University of Oslo
Department of Informatics

Master Thesis

Combined PIM-PSM

Sergei Savenko

19th April 2004

Preface

This thesis was written as a part of a master degree at the University of Oslo. The work has been carried out in the period January 2003 and April 2004.

This thesis might be interesting to anyone interested in the alternative approaches to software development. The Model Driven Architecture (MDA) defines one of such approaches. It allows to separate the system functionality specification from its implementation on any specific technology platform. Following the MDA approach, the first step of a software development project is the creation of a Platform Independent Model (PIM). Such PIM can then be mapped to one or more Platform Specific Models (PSMs). Keeping a PIM and its PSMs separate has many advantages, but it also leads to some problems. One of them is the synchronization of several models. Whenever a PIM is updated, all corresponding PSMs must also be updated to reflect the changes. A solution could be to combine a PIM and all its PSM in one and the same model. How this can be done is the main question of this thesis.

I would like to thank my supervisor professor Birger Møller-Pedersen for encouraging me in this project, reading the text, and providing timely and valuable feedback.

Oslo, April 2004
Sergei Savenko

Contents

1 Introduction **6**
1.1 What this thesis is about? 6
1.2 Getting started. 10
1.2.1 Kinds of software 10
1.2.2 Layering of enterprise applications. 11
1.2.3 Choice of technology. 13
1.2.4 Choice of example. 13
1.2.5 Introduction to the Pet Shop sample application. 15

2 Model Driven Architecture **18**
2.1 Role of MDA in Software Architecture 18
2.2 The Meta Object Facility 19
2.2.1 MOF meta-models 20
2.2.2 MOF meta-data 22
2.3 MDA components 22
2.3.1 Classification of models 22
2.3.2 Operations on models 24
2.3.3 Relations between models 24
2.4 PIMs and PSMs 25
2.4.1 PIM 25
2.4.2 PSM 27
2.5 Modeling languages 28
2.5.1 UML 28
2.5.2 Extending UML 32

3 Platform Independent Models **35**
3.1 Design principles of PIMs 35
3.1.1 Describing classes, attributes, and methods 36
3.1.2 Describing relationships 38
3.2 Pet Shop PIM 39
3.2.1 UML profile for Pet Shop Application 40
3.2.2 Customer module 41
3.2.3 Catalog module 44
3.2.4 Order module 46

4 J2EE versus .NET platform 49
4.1 Introduction 49
4.2 Runtime environment 52
4.3 Java vs. C# 54
4.4 Comparison of enterprise application architectures 59
4.4.1 Presentation logic tier 60
4.4.2 Domain logic tier 61
4.4.3 Datasource logic tier 64
4.5 Summary 66

5 Introduction to model transformations 70
5.1 Role of model transformations in MDA 70
5.2 Basic concepts of model transformations 73
5.3 Overview of the existing approaches to model transformations . . 76
5.3.1 Main features of model transformation approaches 76
5.3.2 Overview of model transformation approaches. 77

6 MOF-based PIM-to-PSM transformations 82
6.1 General concepts 82
6.2 Model type mappings 84
6.3 Model instance mappings 95

7 Combining PIM and PSM 116
7.1 What is a *Combined PIM-PSM*? 116
7.2 Is it possible to combine PIMs and PSMs? 117
7.2.1 Requirement 1: Existence of standardized and predefined languages for PIMs and PSMs 118
7.2.2 Requirement 2: Existence of expressive and unambiguous mapping rules 121

8 Conclusions and future work 124

Chapter 1

Introduction

> The entire history of software engineering is that of the rise in levels of abstraction.
>
> — *Grady Booch*

1.1 What this thesis is about?

The non-stopping technological progress in different areas of computer science has brought a lot of new opportunities to software development. Many of the problems that we can solve now were almost impossible just few years ago. The hardware becomes cheaper and more efficient every day, thus enabling us to create programs that need high computational power. Advances in networking has led to the broad acceptance of distributed systems. The Internet enables users throughout the world to access distributed services wherever they may be located. Not only computers need software nowadays. Many electronic consumer devices have software that can connect them to Internet and perform various tasks. We can see that software development is no longer limited to a small amount of businesses and research institutions, but it can serve us in all parts of our lives.

The ability to write highly functional programs does not imply that it is an easy task, however. The price we must pay for that ability is the increasing complexity of the software development process. As the complexity of the system gets greater, the task of building the software gets exponentially harder. It is not unusual nowadays that the source code for some programs is tens or hundreds of thousands of lines. It is obvious that such code is very difficult to comprehend and maintain. In many situations (nuclear reactors, flight controllers, etc.) even small failures in software are totally unacceptable.

Another problem of large programs is that it takes a lot of time to port them from one technology to another. This problem is crucial because all new technologies tend to become obsolete and forgotten due to the continuous appearance of other technologies. As an example of this we can consider large COBOL programs

that were written in the 80's for the banking sector and are still in use today. It is now desirable to convert them to some modern technology but it is a very expensive task.

The successful software development project depends on many things. Three of the most important aspects that must be taken into consideration are: cost of production, quality of the produced software, and longevity of the produced software. They may be referred to as viability variables of the software development. We can expect to rise the quality of the entire project only by improving qualities of all three variables at the same time. We should avoid sacrificing some variables in order to improve the others but it is not so easy in practice.

It is not hard to see what risks the complex and large projects can face. The cost of production depends mainly on the time which is used on the project. Writing thousands of lines of code is a very time-consuming task even if some part of code can be automatically generated by IDE tools. Thus, the price of the project increases. Quality relates to the absence of all logical errors in the software in order to ensure its intended behavior. Chances that some errors would occur in complicated projects are rather big. The longevity addresses code portability and code reuse. It is unacceptable to use a lot of time to develop some software and to use that software just for a short period of time.

We can see that without some smart techniques we should not be able to take advantage of all opportunities that the technological progress can offer us. It is not the first (and certainly not the last) time we meet this problem and it was always solved by using the same approach: by raising the abstraction level in software development [5].

At the early age of computer industry the programs were written in machine codes. People soon realized that this was a very inefficient method of programming. Writing programs with '0's and '1's needed enormous efforts from programmers and development of large systems was impossible. Assembly languages added a level of abstraction to programming by introducing human-readable commands. The programs became much shorter and easier to understand. Fewer errors were made. However, the common problem of machine languages and assembly languages is that they were hardware centric. It means that they issued commands directly to hardware components. A program written for one machine was not easily portable to another machine.

3GL languages added another level of abstraction. The commands became even closer to human languages by encapsulating processing logic. The simple PRINT command had the same effect as tens of lines of code in assembly languages. 3GL languages offered also structuring constructs which reduced the amount of hand-written code. The major benefit of 3GL was portability. The same language could have many different compilers (e.g. C) and once written program was no longer tied to a specific machine. Another important achievement of 3GL was the ability to call OS commands. This removed hardware details from high-level programming by moving them to the OS's area of concern. The effect of 3GL was the significant improvement of all three viability variables.

Another step in the raising of abstraction in software development is the appearance of middleware and virtual machines. Middleware offers common services which are independent of different operating systems and platforms. Virtual machines sit on top of OS and hide its special low-level features from the programmer.

We can see that the complexity of software increases as the technology evolves and to deal with this complexity new abstraction levels must be added to the development process. Many people believe that the next step in this direction is declarative programming. Declarative programming languages define what the program should do, but not how the result should be achieved. The only thing the programmer has to do is to declaratively specify the intended behavior of the system and all of the actual implementation should be generated automatically. This is not a completely new idea and some efforts have already been made. One example is 4GL. The main disadvantage of 4GLs is that they have a very limited area of application. They are used mostly in GUI programming. However, in this specific area they perform reasonably well.

Modeling is the natural alternative to declarative programming. It is used in many phases of software development process which include: *business modeling*, *requirements modeling*, *architecture modeling*, *database design modeling* and many more. It is popular to build models because they represent a simplification of reality and help us to analyze complex systems that we cannot comprehend in its entirety. The main four aims that we can achieve through modeling are[3]:

1. Models help us to visualize a system as it is or as we want it to be.

2. Models permit us to specify the structure or behavior of a system.

3. Models give us a template that guides us in constructing a system.

4. Models document the decisions we have made.

Despite of many obvious advantages of using models, the current state of modeling has some limitations. Most of models which are created during the system development process are totally independent of each other. Different models are used to describe different aspects of the system (e.g. structure, functionality, behavior, QoS) and it is very hard to trace correspondences between them. We can neither develop a single model which describes a whole system in its entirety nor can we combine separate independent models in such single model. Therefore, we cannot develop models which can contain enough technical information for automatic code generation.

Many people believe that these limitations can be overcome. One of possible approaches to model-based system development is called Model Driven Architecture (MDA). It was proposed by Object Management Group (OMG) in early 2002. The main idea of MDA is to develop a new model at each step of the software development process. These models can describe different parts and aspects of the system but they all must follow two specific rules:

- Every subsequent model must be less abstract than the preceding model. The system development process begins with making a model which describes the system in a very abstract way. The last model should in theory be so detailed that it would contain enough technical details to produce program code.
- Every subsequent model must rely on the preceding model. It should be possible to trace correspondences between every two subsequent models. In such a way we can ensure that all models describe the *same* system, although from different viewpoints and with different levels of abstraction.

One of the characteristics of MDA is that it separates business or application logic from the underlying platform technology by defining Platform Independent Models (PIMs) and Platform Specific Models (PSMs). This allows us to concentrate on the core business logic of the system and not to bother about details that would depend on our choice of the underlying technology. PIMs describe only such structural and behavioral properties of the system that do not vary with the change of the platform. These properties are essentially the business behavior of the system that is expected by the end users. They should always be the same regardless of what actual implementation of that system we may choose. The main advantage of PIMs is that they are often significantly smaller than the usual models because everything that is not related to the business logic is dropped. It is much easier to understand, analyze, develop and maintain small models. Thus, the risk of making some logical errors in PIMs is reduced and in this way PIMs can ensure quality of the software.

When we have a well defined PIM, the next step is to decide which platform we want to use in our software development. Since PIM does not depend on any specific platform (by definition) it can not be used directly without some preparations. Every platform has its own unique technical properties that must be combined with a PIM to make use of that platform. The result of that operation would be PSM. It is possible to have many PSMs for one PIM. It is an important feature of any PIM that we can choose an arbitrary platform and make a PSM without redefining any business rules from the PIM. This increases longevity of our software. If we want to change the underlying platform for our application, the only thing we have to do is to change the PSM according to the new technical details. It is a lot easier task than redefining the whole model.

Experience shows that it is not convenient to have PIM and PSM as two completely separate models. If we, for example, make some changes to a PIM then we must immediately update the corresponding PSM to synchronize these models. The situation gets even worse if we have several PSMs. This problem can be overcome if we manage to keep both the PIM and all its PSMs in the same physical model, but still logically separated from each other. Such approach can be called "Combined PIM-PSM" and it is the title of this master thesis. My goal is to decide whether it is possible to combine PIM and PSM or it is an unrealistic proposal. I will investigate what problems combining PIM and PSM involves and what solutions to solving these problems can exist.

This thesis is intended to be easy to understand for all readers, even for those

who have never heard of MDA. It is structured in the following way:

- In the rest of this introductory chapter I will introduce the Pet Shop application which will later serve as an example for making PIM and PSMs.
- In chapter *Model Driven Architecture* I will cover the most important aspects of the MDA. In particular, I will define PIMs and PSMs more formally and show what modeling facilities MDA can offer.
- In chapter *Platform Independent Models* I will discuss the main issues of PIM modeling and make a PIM of the Pet Shop application.
- In chapter *J2EE versus .NET platform* I will compare J2EE and .NET — the platforms which I will use for modeling PSMs of the Pet Shop application.
- In chapter *Introduction to model transformations* I will discuss general concepts of model transformations. We will see what is a model transformation and what research in this area has been done.
- In chapter *MOF-based PIM-to-PSM transformations* I will try to define mapping rules which can be used for transforming the Pet Shop PIM into two different PSMs — one for .NET and one for J2EE.
- Chapter *Combining PIM and PSM* is the central chapter of this thesis. I will discuss the main concepts of combining PIM and PSM.
- In chapter *Conclusions and future work* I will summarize the main ideas of this thesis and propose the future work.

1.2 Getting started.

1.2.1 Kinds of software

This master thesis is about modeling software with PIMs/PSMs and my first task was to decide what particular kind software I will take as example. There are many different kinds of software such as telecom software, operating systems, virtual reality applications, artificial intelligence systems, database management systems, and many others. Each of them has own challenges and complexities. In VR systems the main problem is complex geometric calculations, while telecom software may have very hard multithreading problems.

The choice of kind of software for my examples should depend on what problems PIMs/PSMs are best suited for. Theoretically any kind of software should benefit from use of PIM/PSM. After some judgment I came to conclusion that the most suitable kind of software for my purposes is enterprise applications. Examples of enterprise applications are payroll, patient records, shipping tracking, cost analysis, insurance, supply chains, accounting, and customer service. Almost all enterprise applications (another term "information systems") share several common properties:

- Enterprise applications usually involve *persistent data*. The data is persistent because it needs to be around between multiple runs of the program. Even if there is a fundamental change and the company installs a completely new application to handle a job, the data has to be migrated to the new application.
- There is usually *a lot of data*. Enterprise applications use often databases that store many gigabytes of data.
- Usually many people *access data concurrently*. Some Web-based systems allow tens of thousands simultaneous users. The problem is to make sure that several people don't access data at the same time in a way that causes errors.
- User interaction is an important part of any enterprise application. *A lot of user interface screens* are needed to handle much data. The data has to be presented lots of different ways for different purposes.
- An enterprise may have many various enterprise systems which are built at different times with different technologies. Very often it is necessary to *integrate* all these systems.
- Even if a company unifies technology for integration, they run into problems with differences in business process and *conceptual dissonance* with the data. The same data may have completely different meaning for different departments in the enterprise.
- Each enterprise system implements some *business logic*, which is an array of sometimes very strange conditions that often interact with each other in surprising ways. Complex business logic is the main reason for difficulties in developing enterprise applications.

We can notice that the core part of any enterprise application is business logic which perfectly fits into a PIM model. Another important observation is that most enterprise applications are intended to live many years (perhaps tens of years) and they are subjects to inevitable technology shifts. This is where PSMs are valuable. The problem of integration of different enterprise systems and conceptual dissonance can also become easier if we apply PIM/PSM techniques. After all, it is not hard to understand why OMG advocates MDA mainly for enterprise applications.

1.2.2 Layering of enterprise applications.

Most of enterprise applications are very large. In order to deal with this complexity they are usually divided into separate layers with distinct responsibilities. Layers are organized in such a way that the higher level uses services of the lower layer, but the lower level is completely unaware of the higher layer. The main benefits of breaking down a system into layers are:

- It is easier to understand a single layer without knowing much about the other layers.

- Each layer can be substituted with alternative implementations of the same basic services.
- Dependences between layers are minimized.
- Layers make good places for standardizations.
- Layers don't depend on higher-level layers. It means that higher-level layers can be freely substituted.

My interest in the structure of enterprise applications is based on the idea that some layers are again better suited for my examples. Of course it should be possible to model each layer with PIM/PSM. One of the goals of MDA is automatic synchronization between different layers and that means that every part of the enterprise application must be carefully modeled. Nevertheless, some layers may be more important then others.

To choose appropriate layer(s) we take a look at how modern enterprise applications are usually organized. Layering of enterprise applications has undergone a long evolution and now it is common to use three primary layers: *presentation logic*, *domain logic*, and *datasource logic*. Different authors use different notations but this is unimportant since the responsibilities of each layer remain the same. Each of these layers can in turn be divided further but we don't consider this fact. Here I summarize these three layers:

- *Presentation logic* is about how to handle interaction between the user and the software. The primary responsibilities of the presentation layer are to display information to the user and to interpret commands from the user into actions upon the domain and data source. The user can be either a human or another program. Presentation logic is the highest layer in hierarchy.
- *Domain logic* is also referred to as business logic. This is the work that involves calculations based on inputs and stored data, validation of any data that comes in from the presentation, and figuring out exactly what datasource logic to dispatch, depending on the commands received from the presentation. Domain logic layer lies under presentation logic and above datasource logic.
- *Datasource logic* is about communicating with other systems that carry out tasks on behalf of the application. For most enterprise applications the biggest piece of datasource logic is a database that is primarily responsible for storing persistent data. Other examples are transaction monitors, messaging systems, other applications, and so forth. Datasource logic is the lowest layer.

Presentation logic layer is currently not very interesting (in my opinion) for PIM/PSM modeling. I can find at least two reasons for that. First of all it is not always clear how to model it with usual UML diagrams. In the case of Web-based applications a big part of presentation logic layer would consist of active

server pages which are provided by many programming languages. Though it is possible to depict "pages" in UML diagrams with the help of stereotypes, those stereotypes contain minimum semantics.

Another reason is that presentation logic is not long-lasting. Sometimes it is changed several times a year to provide different interfaces to the enterprise application's business functionality. Presentation logic layer doesn't provide any functionality to the underlying layers and can be safely removed from the overall model of the application without affecting business logic.

The main purpose of datasource logic layer is providing domain logic layer access to external data resources. This can be done by using some kind of object relational mappings, messaging, legacy systems, etc. It may seem that since datasource logic layer doesn't directly participate in processing that data, its role is not very big. However, that is not so simple. Behavior of domain logic layer is totally dependent on the correctness of the provided data. A little error in the datasource logic layer can corrupt the whole enterprise application. That is why datasource logic layer deserves very careful attention.

The last layer to discuss is domain logic layer. This layer is the heart of every enterprise application because it is here the main business processing occurs. Not surprisingly most of development time is usually spent on domain logic layer. It is the largest and the most difficult part of any enterprise system.

It is important to mention that these three layers are not always fully separated. I discussed an ideal situation where each layer has distinct responsibilities. Reality is usually more complex. Often business logic is spread across several layers. I think that the best way to illustrate how PIM/PSM works is to show examples based on the two lowest layers: domain logic layer and datasource logic layer.

1.2.3 Choice of technology.

After deciding what kind of software I will concentrate on in my work, the next thing to do was to choose two different technologies (platforms) that can be used in developing that software. J2EE and .NET seemed to be the most reasonable platforms because:

- Both .NET and J2EE have gained popularity among software developers and they look like the most common platforms for enterprise application development in the future.
- Both J2EE and .NET provide almost the same functionality. This is important because it only makes sense to create PIM for comparable platforms. If the difference between platforms is big then no PIM that is common for both platforms would be possible.

1.2.4 Choice of example.

Choice of a good example is a very important task. Many articles about MDA are written, but almost all of them concentrate on theoretical questions rather

than on practical examples. This makes it difficult to understand how and where the theory can be used in the real world. Another important point is that description of theoretical issues of MDA without supporting them by examples sounds like an advertising. A reader can get a suspicion that the theory doesn't really work or that nobody has ever tried to apply the theory on the real-world examples.

I will try to illustrate PIM/PSM modeling and model transformations by many different examples. There can be two alternatives for how this can be done. The first alternative is to use many simple examples that are not connected to each other in any way. The advantage of this approach is it's simplicity. The examples can be relatively small and easy to understand. Unfortunately, this approach is not much better than providing no examples at all. Many separate examples can not show how the MDA process can be used during the entire cycle of development of a large software product.

The second alternative is to choose one big application and to use it as an example throughout this master thesis. By doing this I can illustrate many phases of MDA methodology (making PIM, making PSMs, transformations between PIM and PSMs, combining PIM and PSMs) by solving one big problem from a real world. Such a large example can sometimes be broken into several smaller ones but this does not hurt since all these small examples remain to be interconnected to each other and can be seen as a whole.

I have chosen the famous *Pet Shop* application to be an example. There are several reasons for my decision:

- I don't have to invent a completely new example, I can base it on the existing one.

- Many people have heard about the Pet Shop application and it is always easier to work with familiar stuff.

- The Pet Shop application is an example of a large-scale enterprise application that can be met in a real world.

- There already exist two different implementations of the Pet Shop application. The first one is based on J2EE platform (implemented by Sun Microsystems [8]) and the second one is based on .NET platform (implemented by Microsoft [15]).

- Each implementation of the Pet Shop application is meant to be a guideline for developing enterprise applications for the underlying platform. Both implementations use *best practices* and *patterns* that are especially useful for the underlying platform.

- Both existing implementations of Pet Shop applications are designed without considering platform independent issues of the problem domain. In other words, the PIM/PSM methodology doesn't affect the final models.

Figure 1.1: The Java Pet Store and the .NET Pet Shop

1.2.5 Introduction to the Pet Shop sample application.

Here I will give a brief introduction to the Pet Shop sample application. The original application, the *Java Pet Store* was developed by Sun Microsystems to demonstrate the use of Java BluePrints principles in a real application design. The application provides an emphasis on the features and techniques used to show real world coding examples. After appearance of .NET, Microsoft designed the *.NET Pet Shop* application. The idea was to compare performance of the same application built for two different platforms. To make such a comparison reasonable the functionality of .NET Pet Shop was almost identical to that of the Java Pet Store (figure 1.1).

The Pet Shop application is an e-commerce application where customers can by pets online. As any other enterprise application it is very large and it actually consists of four separate sub-applications that cooperate to fulfill the enterprise's business needs.

Pet Shop e-commerce Web Site is a Web application which shoppers use to purchase merchandise through a Web browser.

Pet Shop Administration Application is a Web application that enterprise administrators use to view sales statistics and manually accept or reject orders.

Order Processing Center is a process-oriented application that manages order fulfillment by exchanging data in XML format with other sub-applications by means of messaging middleware and sending acknowledgements to the customers by email.

Supplier is a process-oriented application that manages shipping products to customers and maintains product database.

The whole Pet Shop application is very big and this master thesis doesn't have enough place to consider every part of it. I have decided that it would be sufficient to look at just one part of the application — the Pet Shop e-commerce Web site. Another important assumption is that I am not going to use a precise copy of the original application. The main reason for this decision is that the

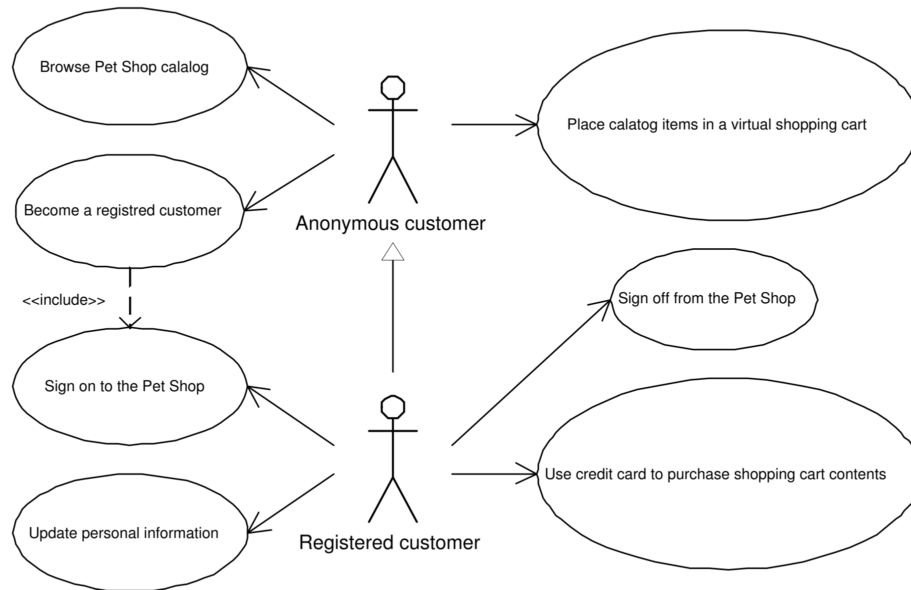


Figure 1.2: Use Case Diagram for the Pet Shop e-commerce Web application

functionality of the .NET Pet Shop is not completely identical to the functionality of the Java Pet Store. Another reason is that there have been developed several versions of both Java Pet Store (current version 1.3.2) and .NET Pet Shop (current version 3.0) all of which slightly differ in the functionality. New versions are now under development. Due to all these observations I will "invent" my own Pet Shop application that lies somewhere in the middle between the original versions.

The functionality of the Pet Shop e-commerce Web Application (I will call it later just *Pet Shop*) is best explained by looking at figure 1.2 which is the Use Case diagram. All users of the application can be divided into two categories — *anonymous* customers and *registered* customers. The main difference between them is that only the registered customers can submit an order for chosen products.

All customers can browse the product catalog and add products to the shopping cart. The products in the product catalog are divided into several categories such as *Birds*, *Reptiles*, *Fish*, and others (this can be seen at figure 1.1 on the preceding page). The customer can choose a category and get a list of all products in the category. For example there can be *Bulldog* and *Chihuahua* in the *Dogs* category. Each product can be explored further for the product items. The product item (e.g. *Adult Male Chihuahua*) can be put in the shopping cart.

Only registered customers can purchase shopping cart contents. Unregistered customers must register themselves in the Pet Shop application in order to be able to purchase anything. They must provide personal information that in-

cludes *contact* information, *credit card* information, *profile* information, *billing* information, and *shipping* information. The registered users can update personal information at any time.

Chapter 2

Model Driven Architecture

> MDA is an approach to system development, which increases the power of models in that work. It is *model-driven* because it provides a means for using models to direct the course of understanding, design, construction, deployment, operation, maintenance, and modifications.
>
> — *from MDA specification*

This chapter presents some of the main ideas of Model Driven Architecture (MDA) promoted by Object Management Group (OMG). The work on defining MDA is far from being finished and not all aspects of MDA are standardized and adopted by the industry. We will look at the concepts that base the foundation of MDA and that will be frequently used throughout this document. For further details on MDA see [5], [13], [11].

2.1 Role of MDA in Software Architecture

MDA is an architecture definition framework for system architecture development methodologies. In other words, MDA describes which architectural aspects should be described and how these aspects should be represented. Many different architectures development methodologies can exist, but in order to be called MDA-compliant they all must admit to some specific rules defined by MDA. These rules describe what kind of artifacts will constitute a concrete architecture, how these artifacts are related and guidelines for how these artifacts should be constructed.

OMG (Object Management Group) presents MDA as a standard framework for designing software applications. This framework captures commonalities of various platforms and by doing this it provides following benefits:

- Applications based on different platforms can interoperate with each other.
- The lifetime of software increases.
- MDA framework remains flexible in face of constantly changing infrastructure.

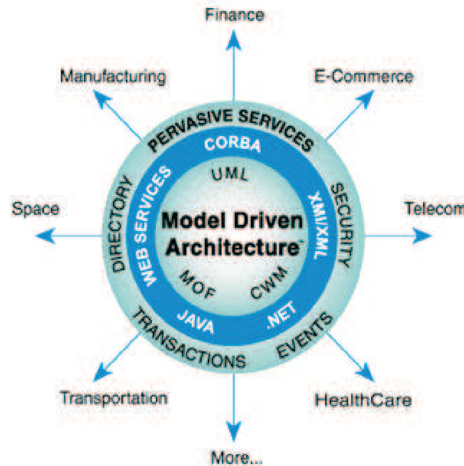


Figure 2.1: Model Driven Architecture

Figure 2.1 diagrams the MDA. It consists of three main parts. The inner circle includes the concepts that are common for various architectures on the marked. This circle contains key MDA technologies that should be used in all MDA-compliant methodologies. These technologies will be explained later in this chapter. The next ring contains specific platforms that are current targets for MDA. They include Web Services, CORBA, J2EE, .NET and XML. The last circle is the pervasive services (directory, security, distributed event handling, transactionality, persistence, and other services). These services are common for all enterprise applications regardless of what platforms they are based on. The arrows that point in different directions show that MDA can be used in many markets.

2.2 The Meta Object Facility

Very often a combination of several different technologies is used in a single software development project. A good example is a combination of relational database and a program written in a specific programming language which uses the data from that database. We can extend this example if we suppose that the underlying technology for our project is J2EE or .NET. In this case we also need some XML descriptors which are quite different from the original programming languages. We can go even further and suppose that we also use CORBA IDL to expose some components as CORBA objects. The MDA principles dictate that every part of our system should be presented as a model. Since different technologies have different special features, it is obvious that the models of different parts of the system would also be quite different. It means that after modeling the whole system we would sit with a bunch of models that would eventually have very little in common. Despite of the fact that the whole system may consist of several parts that are based on different technologies, it is always convenient to see such system as one entity. It is only possible if we can easily integrate all the models that we acquire in the modeling phase of our

project.

Every model is written in specific language that is powerful enough to provide all the constructs that are necessary for building that model. Such languages are called meta-models. Many meta-models can exist to suit different modeling demands. A meta-model acts as a filter to extract some relevant aspects from a system and to ignore all other details. This implies that models of different parts of a system (e.g. database schema, XML descriptors, OO component architecture) would be based on different meta-models.

It can be very difficult to integrate models of different parts of a system because these models can be based on different meta-models which may have unique sets of constructs. There has been an effort to design a common meta-language that would contain all the necessary constructs. Such meta-model is, however, impossible. The reason is that it isn't realistic to expect that one language can describe all aspects of a system (aspects that exist now and aspects that may appear in the future).

2.2.1 MOF meta-models

MDA solves this problem by introducing the Meta Object Facility (MOF) which is a meta-meta-model. A meta-meta-model is a language for defining meta-models. Instead of trying to design a language that would contain all modeling constructs, OMG designed a language which is capable to define these constructs. That is the purpose of meta-meta-model. It can be easily used to describe constructs used by relational data models, UML class models and many more.

There are several possibilities to define a meta-meta-model. Usually the definition is self-reflexive, i.e. the meta-meta-model is self defined. In other words, the MOF uses the MOF to describe itself. The MOF defines a MOF-compliant model of its own constructs. A meta-meta-model is based at least on three concepts (entity, association and package) and a set of primitive types. The OMG MDA postulates the use of MOF as a unique meta-meta-model for all IT-related purposes. The MOF contains all universal features, i.e. all those that are not specific to a particular domain language. Among those features we find all that is necessary to build meta-models and to operate on them.

Maintaining a specific tool for the MOF would be costly, so the MOF is aligned on the core part of one of its specific meta-models: UML. UML thus plays a privileged role in the MDA architecture. As a consequence, any tool intended to create UML models can easily be adapted to create MOF meta-models. The MOF meta-models look like UML class models where modeling constructs are modeled as classes and the properties of the construct as attributes of the class. Relationships between constructs are modeled as associations. It should be mentioned that although the MOF's constructs are object-oriented (since they are borrowed from UML), they can be used to define non-OO meta-models.

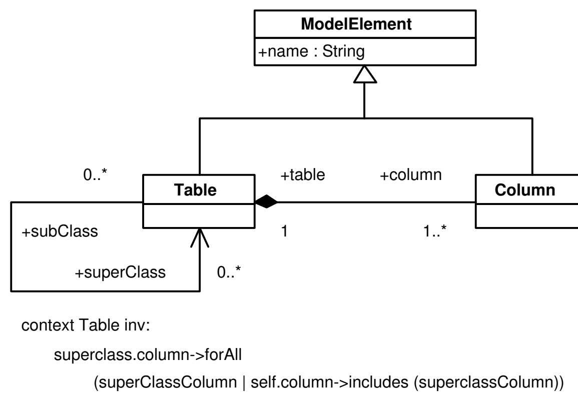


Figure 2.2: Using MOF to define a simple meta-model

A meta-model is composed of three parts: terminological, assertional, and pragmatics. The terminological part corresponds to UML class diagrams. The assertional part corresponds to OCL assertions that may decorate the various elements of meta-model. The pragmatics corresponds to details that could not fit into previous parts. Example of a pragmatic item is for example how to draw some particular concepts or relations. Usually the pragmatics elements are expressed in natural language informal descriptions.

Figure 2.2 shows en example of a simple meta-model that defines elements: *Table* and *Column*. This meta-model defines that a table can have several columns and that both tables and columns have attribute *name* of type *String*. In addition to that it also specifies that a table can have superclasses and subclasses. An OCL statement declares that a table inherits the columns of its superclasses.

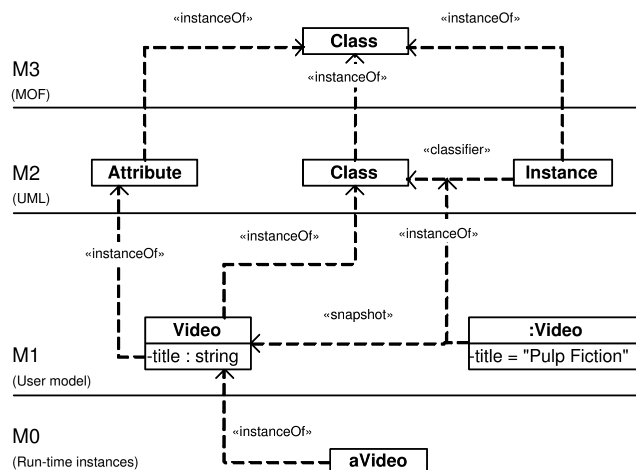


Figure 2.3: MOF meta-levels

MOF architecture consists of four "meta-levels". They are named M3, M2, M1 and M0. Figure 2.3 on the page before illustrates their use. Here is the short description of all levels.

M3 is MOF, whose elements are the constructs MOF supplies for defining meta-models. These elements include *Class*, *Attribute*, *Association*, and so on.

M2 is populated by meta-models defined via the MOF constructs. Example of such meta-model is *UML meta-model*.

M1 is populated by models that consist of instances of M2 constructs. An example is some class model which is modeled in UML.

M0 is populated by objects and data, which are instances of M1 elements.

2.2.2 MOF meta-data

It was mentioned earlier in this chapter that a single software project may contain many different types of models. All these models describe some underlying data and thus they can be called meta-data which means "data about data". Meta-data represent different software development artifacts and it is very important to be able to manage them in a common uniform way. This means that it should be possible to have some meta-data management facility to manipulate meta-data, that is, to create new meta-data in the repository, read meta-data already in the repository, update meta-data in the repository, delete meta-data from the repository and interchange meta-data between different repositories. Since all meta-data (models) are based on the MOF, that should not be a problem.

The MOF contains some features to serialize models and meta-models in order to provide a standard external representation which is necessary for achieving the meta-data management goals. The serialization is accomplished by industry-standardizes mappings of the MOF to specific middleware, 3GLs and information formats. These mappings allow automatic transformation of meta-model's abstract syntax into concrete representation based on XML DTD, XML Schema, Java, and CORBA technology. It is possible to develop mappings to other technologies as well.

2.3 MDA components

This section describes different types of models (and meta-models) and properties associated with them. MDA suggests many different kinds of models which differ in a variety of properties [2]. All these models are designed to serve specific purposes but in many ways they are closely related to each other.

2.3.1 Classification of models

Some model classification criteria are presented here. However, not all of them are used equally frequently:

- Every model describes either some *products* or *processes*. Product models describe a structured set of artifacts produced or consumed. Typical examples of product models are such UML models as component diagram, class diagram or deployment diagram. Process models characterize tasks, roles, actors, goals and other behavioral properties of the system. Examples are business models, requirement models, UML use case models, etc.

- Models can be *static* or *dynamic*. A static model is invariant while a dynamic model changes over time. There can be static models of dynamic systems.

- Models can be *executable* or *non-executable*. Every program's source code can be considered to be an executable model constructed with a very low degree of abstraction. UML models are usually non-executable.

- A model may be *atomic* or *composite*. An atomic model contains only basic elements while a composite model contains at least one another model.

- Models may be *primitive* or *derived*. A derived model may be obtained from other models (primitive or derived) by a derivation operation or a sequence of such operations. A derivation operation is a simple and deterministic model transformation.

- A model may contain *functional* and *non-functional* elements. Typical non-functional elements are related to QoS properties like performance, reliability, security, confidentiality, etc. The elements of a non-functional model are usually related to specific elements in a base functional model.

- An *essential* model is a model that is intended to stay permanently in the model repository system. A *transient* model is disposable, i.e. it has been used for some temporary purpose and has not to be saved.

- When we have a conversion to be done between an important number of different meta-models, it may be interesting to define a *pivot* meta-model to facilitate the transformation process.

- UML is a product meta-model for *object-oriented software artifacts*. Models of *legacy* systems differ from them in many ways.

- A *source program*, written in a given program language is also a model. The meta-model corresponds to the formal grammar of the language and the model is executable. One particular execution of this program may also be considered as a system and other models can be extracted from this execution (e.g. *execution trace*).

- Two other important types of models are *Platform Specific* and *Platform Independent* models. Distinctions between PIMs and PSMs are covered in section 2.4 on page 25. We will refer to these two kinds of models many times in this document.

2.3.2 Operations on models

Operations that can be performed on different types of models can also be classified according to several parameters. Some operations are specific to certain types of models and others can be used on various types of models (sometimes even on combination of them).

- *Monadic* operations apply to a single model and *dyadic* operations apply to two models. There are many apparently monadic operations that turn out to be dyadic, if we are able to define the argument as a model or a meta-model.
- A model may be *built*, *updated*, *displayed*, *queried*, *stored*, *retrieved*, *serialized*, etc. These operations suppose the existence of explicit or implicit meta-model such as UML. All the above operations can be applied to meta-models as well.
- It must be possible to efficiently *store* and *retrieve* models to/from persistent storage. In many cases using simple file systems after XMI serialization lacks efficiency and don't scale up.
- *Filtering* a model means extracting a specific view on this model. The important question here is how the view is specified and if this operation may be considered as a dyadic operation producing another model.
- Rapid *prototyping* associates some limited executability to a model in order to interactively evaluate its properties before transforming it into an executable system.
- *Code generation* operation may convert a UML model into Java or C# code. Bidirectional transformation may be desirable.
- *Confrontation* operation may produce the similarities and differences between to models, possibly as a new model. An *alignment* of two models would produce the new model if some equivalence rules are defined. A *fusion/merge* of two model is also possible.
- Model *transformation* operation is a central aspect of MDA and it will be discussed in greater detail in chapter 5 on page 70.

2.3.3 Relations between models

Models can relate to each other in a variety of ways. Relations between models can also be presented as models. The two most common of kinds of model relations are:

- *Refinement relations* show how pairs of models describing the same system but at different levels of abstraction relate to each other. Refinement is the converse of abstraction.
- *Correspondence relations* show what correspondences may hold between different models. The correspondences are not always between couples of elements and they are strongly typed.

An important case of correspondence relation is *viewpoint correspondence*. It shows correspondences between pairs of models from different *viewpoints* and unrelated by refinement. A viewpoint on a system is a technique for abstraction using a selected set of architectural concepts and structuring rules, in order to focus on particular concerns within that system.

2.4 PIMs and PSMs

One of the most fundamental issues of MDA is the distinction between specification of functionality and specification of how this functionality is implemented using a particular technology. This distinction is achieved by separating platform independent and platform specific models.

Platform independence is a quality, which a model can exhibit. This is the quality that the model does not call for the support of a platform for a particular type. Like most qualities, platform independence is a matter of degree. The most important question here is what we define as a platform. Generally, a platform refers to technological and engineering details that are irrelevant to the fundamental functionality of a software component. More formally the platform is "a set of subsystems/technologies that provide a coherent set of functionality through interfaces and specified usage patterns that any subsystem that depends on the platform can use without concern for the details of how the functionality provided by the platform is implemented"[13].

Examples of platforms can be: Windows XP, CORBA, .NET and XML. Platforms also have models. A platform model provides a set of technical concepts, representing the different kinds of parts that make up a platform and the services provided by that platform. It also provides, for use in platform specific model, concepts representing the different kinds of elements to be used in specifying the use of the platform by an application. Examples of technical concepts from J2EE platform model are different kinds of enterprise beans. A platform model may itself be thought of as a PIM because it specifies the platform concepts, and not its concrete implementation.

A model is independent of a particular platform if it doesn't use the concepts from the correspondent platform model. A model that doesn't depend on middleware may however depend on OS or programming language. Therefore, we must always specify what platform we mean when we talk about platform independent models. Platform specific models always use some concepts from one or more specific platforms models. This is illustrated in figure 2.4 on the following page. Platforms are usually gathered in platform categories. For example CORBA, J2EE and .NET platforms can be categorized as platforms supporting distributed objects.

2.4.1 PIM

All MDA development projects start with the creation of a Platform Independent Model, expressed in UML. An MDA model will have multiple levels of PIMs.

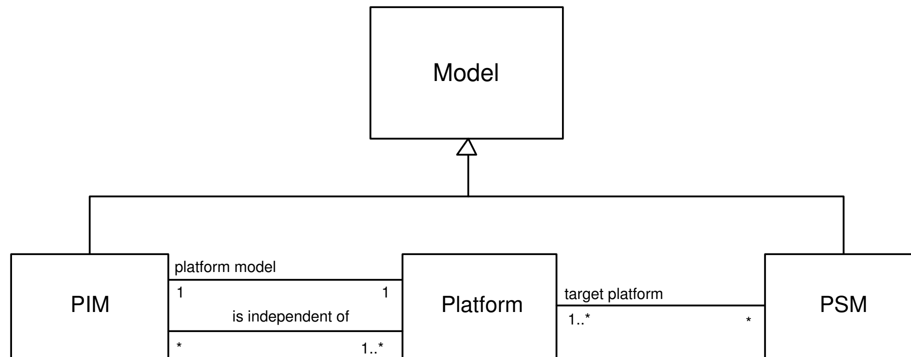


Figure 2.4: PIM and PSM relations

Although all are independent of any particular platform, each except the base model includes platform independent aspects of technological behavior.

The base PIM expresses only business functionality and behavior and sometimes it is just called a "business model". Built by business and modeling experts working together, this model expresses business rules and functionality, as much as possible, by technology. The clarity of this modeling environment allows business experts ascertain, much better than they could if working with a technological model or application, that the business functionality embodied in the base PIM is complete and correct. Another benefit of PIM is that because of its technological independence, the base PIM retains its full value over the years, requiring change only when business conditions mandate.

Many elements of the solution can be incorporated in a PIM as long as they don't refer to a specific deployment platform. For example we can take algorithm hints into account. Sometimes the base PIM is called CIM (Computational Independent Model), which is a PIM where the problem has not yet been worked out a solution.

PIMs at the next level include some aspects of technology even though platform-specific details are absent. Examples that we will see later in this document are taken from J2EE and .NET. Both these platforms have many common concepts related to enterprise applications development. They can include persistence, transactionality, security, some configuration information, and many more. By adding these concepts to the PIM, we enable it to map more precisely to a Platform Specific Model at the next step. PIMs are usually designed in one of a number of OMG-standardized UML profiles. For example, OMG has defined a profile for Enterprise Distributed Object Computing (EDOC) which can be used for modeling collaborations of all types, and a profile for EAI, specialized for applications based on asynchronous communications. Additional profiles are used to define PIMs.

Some of the standard modeling infrastructure that incorporates business behavior in to PIM already exists. One of the examples is OCL (Object Constraint

Language) which can provide a lot of useful information to UML models. It can provide such facilities as :

- *Pre-conditions* and *post-conditions* for operations. A pre-condition must be true when the operation is invoked. A post-condition must be true when the operation is finished executing.
- *Invariants.* An invariant is an assertion about the state of a system that must always be true.

However, OCL has disadvantages. Some of its expressions are very complex and quite often it is much easier to use informal language to describe constraints without loss of precision. Many things can not be expressed in OCL at all.

One of the advantages of PIMs is the ability to specify a system's quality. Quality of the software relates to such aspects as:

Effectiveness - the capability of the software product to enable users to achieve specified goals with accuracy and completeness in a specified context of use.

Productivity - the capability of the software product to enable users to expend appropriate amount of resources in relation to the effectiveness achieved in the specified context of use.

Safety - the capability of the software product to achieve acceptable levels of risk of harm to people, business, software, property or the environment in a specified context of use.

Satisfaction - the capability of the software product to satisfy users in a specified context of use.

Quality covers system performance, which is expected by the end users, rather than system operations. Carefully described software product's quality in a PIM may have a major impact on the actual later implementations. It helps to identify such extra-functional properties of a system as: suitability, accuracy, security, interoperability, fault tolerance, recoverability, understandability, learnability, attractiveness, operability, resource utilization, time behavior, stability, maintainability, adaptability, modifiability, portability, and many more.

Different techniques can be used to describe system's quality. One of them is Component Quality Modeling Language (CQML), which is a lexical language for Quality of Service (QoS) specification. Another possibility is the use of UML profile for QoS.

2.4.2 PSM

Once the first iteration of PIM is complete, it is stored in the MOF and it is ready for transforming into PSM. A platform-specific model is a computational model that is specific to some information-formatting technology, programming language, distributed component middleware, messaging middleware, or some other technology. It combines the specification in the PIM with the details that specify how that system uses a particular type of platform. To produce

a PSM we have to select a target platform or platforms for the modules of the application. The target platform can be for instance: EJB, .NET or Web Services (usually from the same category). During the transformation step, the run-time characteristics and configuration information that is designed into the application model in a general way (PIM) is converted to the specific forms required by the target platform.

According to OMG's MDA definition document, four different ways exist to transform a PIM to a PSM. In increasing level of sophistication and automation they are:

1. A person performs the transformation completely by hand, working each application *ad hoc* without reference to others.
2. A person performs the transformation using established patterns to convert from the PIM to a particular PSM.
3. The established patterns define an algorithm which is implemented in an MDA tool that produces a skeleton PSM, which is then completed by hand,
4. The tool, applying the algorithm, is able to produce the entire PSM.

2.5 Modeling languages

Both PIMs and PSMs are modeled with some specific modeling language which helps to visually illustrate the architecture of a desired system. The choice of modeling language is not restricted. The only condition that a modeling language must satisfy is that it must be powerful enough to express all intended properties of the system. The modeling languages are defined with the help of MOF meta-models. If it happens that none of the existing modeling languages can be used, it is possible to define a new modeling language that will meet additional needs.

2.5.1 UML

In most cases the Unified Modeling Language (UML) is the right language to use. The UML was originally developed by Rational Software Corporation in the mid 1990's as a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software system. Later the UML was standardized by the OMG. The current version of UML at the time of writing is 2.0. Now the UML forms the foundation of MDA and it is heavily used through the entire cycle of any MDA-compliant architecture development methodology.

The Unified Modeling Language provides means for modeling systems at different abstraction levels and to describe system's properties from different viewpoints. UML defines twelve types of diagrams, divided into three categories:

Structural diagrams represent static application structure. They include *Class diagram*, *Object diagram*, *Component diagram*, and *Deployment diagram*.

Behavior diagrams represent different aspects of dynamic behavior. They include *Use Case diagram*, *Sequence diagram*, *Activity diagram*, *Collaboration diagram*, and *Statechart diagram*.

Model Management diagrams represent ways the application modules can be managed and organized. They include *Packages*, *Subsystems*, and *Modules*.

In this thesis we are mainly interested in structural diagrams, and especially class diagrams. When I discuss UML in this section I will have that type of diagrams in mind most of the time. However, many of the ideas can be equally applied to all other types of diagrams.

The MOF meta-model of the UML defines all UML's concepts in terms of abstract syntax and semantics. These concepts can be expressed in a variety of notations which may include Human-Usable Textual Notation (HUNT), XMI, Java objects, CORBA objects, and others. It is even possible to encode UML's properties with own constructs which are based on some concrete technology. However, the most common way to express UML concepts is to use standardized graphical notation.

The UML meta-model is based on the principles of object-orientation and the entire meta-model is represented as a hierarchy of related meta-classes. These meta-classes model different concepts of UML. It follows that UML's concepts

- have names
- can have properties (both simple properties and properties that are other concepts from UML)
- can inherit properties from other UML's concepts

At the top of concept hierarchy lies a simple *ModelElement* concept that has a single *Name* property. All other concepts such as *Class*, *Attribute*, *StructuralFeature*, *BehavioralFeature*, and many others are derived from top concept. Most of the ideas from object-orientation can be found in the UML meta-model. For example, the *Feature* concept has a property named *visibility* which supports encapsulation of class members.

MOF's graphical notation uses UML. This means that concepts from UML meta-model and concepts from UML models are graphically modeled in a similar way. The usual representation of a model concept is a rectangle that is horizontally divided into three parts. The name of the model's concept is located in the upper part of the rectangle. The properties of the model's concepts are placed in the central part of the rectangle and all methods associated with the model's concept lie in the lower part of the rectangle.

Figure 2.5 on the next page shows an *Account* concept from the Pet Shop model. This concept is used to model information which is associated with individual customers. We can see that all properties of the account concept are encapsulated. A user of account can access properties only via special *getter*

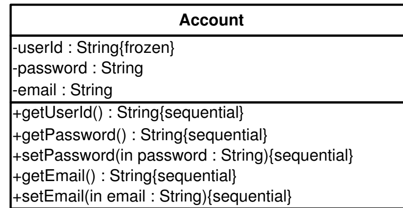


Figure 2.5: Graphical notation of a concept from UML diagram

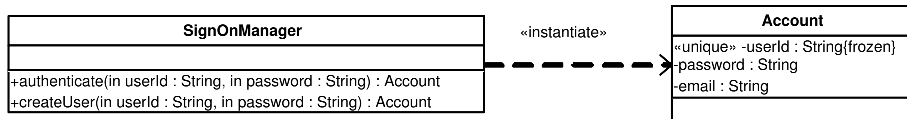


Figure 2.6: Dependency relationship

and *setter* methods. The standard graphical notation of visibility of properties and methods in UML is + (public), - (private), and # (protected).

Concepts in the UML diagram are rarely isolated from other concepts. Most of the concepts are usually related to one or more other concepts in different ways. The three most important kinds of relations are

Dependency is a relationship that states that a change in a specification of one concept may affect another concept that uses it, but not necessarily the reverse. Graphically, a dependency is rendered as a dashed line, directed to the concept that is depended on. Figure 2.6 shows that the *Account-Manager* concept depends on the *Account* concept. This dependency is of kind *instantiate*. *AccountManager* is used to sign in customers by verifying provided userId and password and to create accounts for new customers. Both these operations return an object of *Account* type. Changes in the specification of *Account* can influence specification of *AccountManager*.

Association is a structural relationship that describes how different concepts are connected to each other. An association has a name, role at each end of association and the multiplicity of each end of association. An association relation can also specify navigability at each end. Navigability is shown with arrows. A special kind of association relation is *aggregation*. Aggregation relation between two concepts means that one concept is a logical part of another concept. Aggregation semantics is depicted by a diamond at the end of relation. Figure 2.7 on the next page shows that *Account* has a private property named *creditCard* of type *CreditCard*. This property has multiplicity 1. *creditCard* object can only be a part of some account (can not exist alone) and it can be accessed only via that account.

Generalization is a relationship that describes inheritance. Inheritance can be both single and multiple. Generalization is rendered as a solid directed line with a large open arrowhead, pointing to the parent. Figure 2.8 on

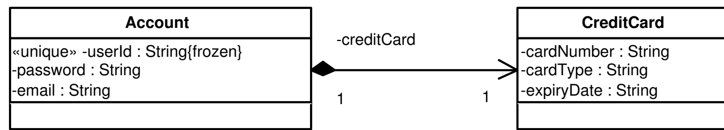


Figure 2.7: Association relationship

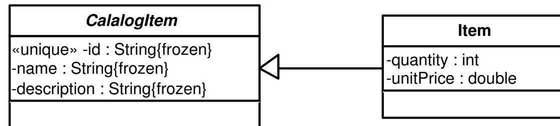


Figure 2.8: Generalization relationship

the facing page shows an abstract class *CalalogItem* and an *Item* class that subclasses it. In this example *Item* inherits 3 properties from its superclass and defines 2 new properties.

Designers of UML have included many frequently used modeling constructs in UML. It is possible to model complex diagrams using only predefined elements of UML. All main aspects of object-orientation such as classes, interfaces, attributes, methods, accessibility modifiers, inheritance modifiers can be found in base UML. Large enterprise applications usually consist of many components and UML allows specifying how these components are working together.

The UML 2.0 adds some new features that provide further modeling possibilities. The OCL will get its own meta-model and it will be possible to express model restrictions in a formally defined way. Action Semantics UML Extensions let modelers express actions as UML objects. An *action* object may take a set of inputs and transform it into a set of outputs, or may change the state of a system, or both. Another good thing with UML 2.0 is that it will allow diagram interchange. It is very important that different modeling tools are able to exchange diagrammatic information in a uniform way. This will certainly raise the value of UML modeling.

While UML has many good qualities, it also has a number of limitations that affect its role in MDA. One of the weaknesses of the UML is the poor separation of concerns in UML meta-model. There are many interdependencies between different packages in UML meta-model and thus it is difficult to use one part of UML without using many other unnecessary parts UML. Another problem with UML is that it takes a long time to standardize it. It means that UML can't immediately reflect all advances in software development. For example, modelers had to wait several years for a well-defined concept of a component in UML.

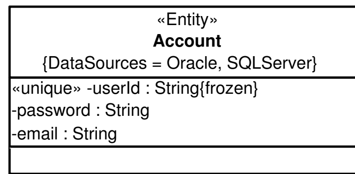


Figure 2.9: Stereotypes and tagged values

2.5.2 Extending UML

The UML meta-model provides many useful constructs that can be used in UML models but like any other modeling language it can not cover all aspects of modeling. Sometimes modelers come to a situation when no predefined UML constructs are suited for their purposes. Within a certain context as an organization or a specific domain the UML is normally used in a specific manner. This special use of the UML has the following aspects:

- A limited subset of the UML concepts is used.
- The models are interpreted in a certain specific manner.

UML solves this problem by allowing modelers to extend UML and add some new constructs beyond those that the base UML defines. UML has three built-in extension mechanisms: *stereotypes*, *tag definitions* and *constraints*.

Stereotypes define some contracts that a component from UML diagram must support and that don't exist in the core UML. UML denotes the assignment of a stereotype to a model element via what UML calls a *keyword*. Keywords are surrounded by guillemets. Figure 2.9 shows an example of an *Account* class. This class is supposed to represent a piece of data that is stored in some external datasource. Any changes made to the object of this class should be permanent, i.e. they should not be missed when the program terminates. It is a common practice to mark such data-oriented classes with «Entity» stereotype. This stereotype doesn't say anything about how the persistency mechanism should be implemented. As we shall see actual implementations for different platforms (J2EE and .NET) can be quite different.

Class is not the only element from UML metamodel that can be extended. We can also extend all other elements including *Attribute*, *Association*, *Operation*, etc. Figure 2.9 demonstrates *unique* stereotype applied to *userId* in *Account* class. It is also possible to define an icon for a stereotype, and to represent a stereotyped model element via the icon.

Stereotypes are sometimes referred to as *lightweight extensions* and may be considered a restricted form of a metaclass (a class that specifies another class). Although both a metaclass and a stereotype can define a new type of model element, a metaclass can have both attributes and associations, but a stereotype can have neither.

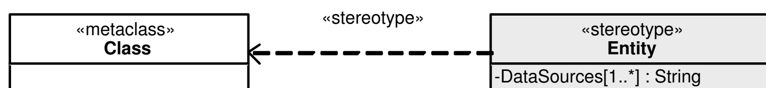


Figure 2.10: Stereotype specification

The definition of a stereotype can include the definitions of tags and constraints. Tags are delimited by curly braces, and the syntax inside the curly braces is *<tag name>* = *<tagged value>*. For example, our definition of «Entity» stereotype defines a tagged value that specifies what datasources can supply data. The purpose of this tagged value is to provide a transparent access to data. The end users should not be concerned about where the data has originated. In figure 2.9 on the preceding page there are two datasources: Oracle and MS SQLServer. Though both datasources in our example are databases, they can also be XML files, legacy applications, or others. We could also define constraints on stereotypes. In the above example we may want to ensure that the list of datasource names is not empty and that it doesn't contain duplicates.

There can also be tags that are not part of any stereotype definition. The modeler adds the values of such tags to instances of the UML model elements that the tags extend. *Frozen* property of the *userId* attribute of *Account* class (figure 2.9 on the facing page) is an example of predefined tagged value in the UML profile for MOF. {frozen} is equivalent to {isChangeable=false}.

When a modeler defines a stereotype, he must explain it, either informally or formally. Informally, a modeler can describe the purpose of a stereotype and its intended use in some human-readable textual notation. Stereotypes can also be defined more precisely in a formal manner using UML graphical notation. Figure 2.10 shows a part of the model that formally defines «Entity» stereotype. The stereotype is represented as a class adorned by the keyword *«stereotype»*. The *«metaclass»* keyword adorns the UML metamodel element being extended. Tags associated with a stereotype are represented as attributes of the class representing the stereotype.

Sometimes developers model a particular domain or problem space that needs a large number of extensions. UML 1.4 provides the ability to define a group of extensions as a *profile*. A UML profile defines:

- A subset of the UML model elements.
- Specializations of UML concepts.
- Limitations and specific requirements for the used concepts.
- Extra (meta)attributes that can be added to the UML models.

A UML profile is a definition of a set of stereotypes, tagged values and constraints that extend elements of the UML metamodel. Profiles are themselves

defined as a UML extension. A *«profile»* is a stereotyped *Package* that contains model elements that have been customized for a particular domain or purpose using the extension mechanisms described above. Many organizations define own UML profiles that serve their specific purposes. For example, the Java Community Process released a UML profile for Enterprise JavaBeans that specifies a standard representation for EJB architecture elements in a UML model.

Profiling is not the only way to extend UML. Another way to extend UML is to extend UML metamodel via MOF. UML extensions that use the full power of MOF are sometimes called *heavyweight extensions*. Instead of defining stereotypes a modeler can create new elements in the UML metamodel that subclass existing elements. For example, we could define a new UML metamodel element called *Entity* that subclasses an existing element called *Class*. The main advantage of that approach is that modeler can use the full semantic power of object-oriented class modeling that MOF offers. For example, it is possible to define new associations between metamodel elements. However, the major disadvantage of heavyweight extensions is weak support for them UML modeling tools.

UML is not the only language used in MDA framework. Some kinds of models are fundamentally different from UML models because their modeling constructs don't fit easily into any of the UML modeling paradigms. When creating a MOF metamodel to define the abstract syntax of such modeling constructs, it often doesn't make sense to try to extend the UML metamodel. In such situations it is easier to define a completely new language by means of MOF. This new language would have all necessary modeling constructs that the application domain requires and will not depend on the UML metamodel.

We have discussed several ways of extending UML and even defining new modeling languages. Most of the time a simple lightweight extension of UML will allow us to customize modeling language according to our needs. I will particularly use this technique to extend UML when I define a PIM for the Pet Shop sample application. Different strategies for extending modeling languages can be combined. It is possible to define a UML profile and extend UML metamodel at the same time. Other combinations may also be of interest.

Chapter 3

Platform Independent Models

> MDA can be thought of as a spectrum with business at the top and technology at the bottom. The result of the work done at the top level is one or more PIMs.
>
> *Jon Siegel, OMG's Vice President of Technology Transfer*

In this chapter we will take a closer look at the main principles and techniques of constructing platform independent models. Through the whole chapter we will design a PIM for the Pet Shop sample application. By doing this we can try to apply the best modeling practices that are recommended by the MDA architecture framework. The resulting PIM should form a basis for further transformations to the platform specific models.

3.1 Design principles of PIMs

At the beginning of this chapter I will mention the main criteria that any PIM must conform to. There are some "design rules" that a modeler must be aware of in order to design a PIM that can be later used in MDA transformations. Some of these rules can be broken but this is undesirable because of the risk that not all MDA functionality can be available at a later point. Here I summarize the main modeling aspects that a PIM designer must constantly think about. As we shall see these concepts are tightly interrelated.

- PIM must contain maximum information from the business domain of the application to describe the behavior of future software as precise as possible.
- PIM must not contain any information about specific platforms that the PIM is independent of.
- PIM must take advantage of the rich set of modeling concepts from the UML but it should be done in a cautious way. Any mistakes can lead to incorrect future transformations.
- A good knowledge of MDA process can spare precious modeling time. The MDA tools are capable of automatically inserting some additional information into UML models based on predefined rules.

The last two aspects can be stated in another way. We can say that the modeler must:

- Ensure that the generator does not reject the model.
- Ensure that APIs and information formats that the generator produces are of the highest possible quality. This in turn ensures that generators can produce optimal implementations.

We saw that the class models are composed of classes that have attributes and methods, and relationships that interconnect classes. All these elements deserve careful attention and we will cover some of the most important issues.

3.1.1 Describing classes, attributes, and methods

UML specifies the general syntax for describing exact behavior of classes, attributes, and methods. Here is the summary:

Class

name [(property-string)]

Attribute

[visibility] name [multiplicity] [: type] [=initial-value] [(property-string)]

Method

[visibility] name [(parameter-list)] [: return-type] [(property-string)]

The minimum requirement for all elements in UML diagram is a name. Name is important because it allows to express the meaning of an UML element in a human-readable fashion. Names for the UML elements are usually taken from the business domain of the application. Names should be chosen carefully to avoid any misunderstandings (about the element's role in the model) and name clashes in the further model transformations. The way the name is typed can also bare additional semantics. For example, names of abstract classes and methods are italicized.

Types that are used in UML models can be primitive and complex. Primitive types are modelled using class notation with the apropriate stereotype. When defining a primitive type, it might be necessary to use constraints to specify the range of values associated with this type. Figure 3.1 on the facing page shows how two primitive types from the Pet Shop PIM can be defined. Complex types are modelled as usual classes. Both primitive types and complex types can be either standard or user-defined. Standard types (e.g. *int*, *long*, *char*) have a well understood semantics and are usually present in type libraries for many existing programming languages. User-defined types (e.g. *money*, *Address*, *CatalogItem*,

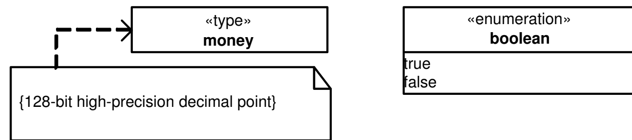


Figure 3.1: Defining primitive types

money) repsesent an absraction of something drawn from the vocabulary of the problem domain or the solution domain.

The important decision that must be taken when specifying standard types in a PIM is what names should be assigned for these types. There are at least two alternatives. If the PIM is designed with some platforms in mind then type names can be taken from the name domain of one of these platforms. The disadvantage of this approach is that PIM can get platform dependency. Another alternative is to use type names that are common across many different platforms (e.g. *int*, *File*) and to invent new names for types that are platform-specific (e.g. *Vector* in Java).

The method signature includes a parameter list which is defined by following syntax:

[direction] name : type [=default-value]

where *direction* can be *in*, *out*, and *inout*. Most of the existing programming languages support these directions of parameter passing.

Definitions of classes, attributes, and methods can include many properties that are predefined by the UML standard. Some of these properties reflect well-known aspects from object-orientation (e.g. inheritence, scope), while others describe the workflow of the program (e.g. call concurrency, threads). In order to preserve encapsulation of attributes and methods it is important to annotate them graphically with visibility modifiers. Attributes should be specified with multiplicities and initial values if necessary. Remembering all these details ensures semantic correctness of the model and reduces amount of later programming work. The same is also true for specifying default values for parameters in method signatures.

Any access to attributes is made via special getter and setter methods. However, it is not necessary to model such methods because this is the work of a MDA tool. The MDA tool needs to know weather attribute is *changeable* or not. The possible values for *changeability* property are *addOnly*, *frozen* or *none*. According to this information the MDA tool will generate only setter method, only getter method, or both. The default value for *changeability* is *none*.

Auto-generation of some class methods is an important aspect of MDA transformations. Generation of accessor and mutator operations is not the only example. Another example is generation *constructor* and *factory* methods. MDA

generation can automatically provide *concrete* classes with these methods. This particularly illustrates the importance of specifying *abstract* attribute of a class.

[5] organizes all operations in two groups: *interesting* and *uninteresting*. Uninteresting operations are implied by the structural features of the class model. Such operations should not be specified in the model because this can lead to code redundancy and unnecessary extra work. Interesting operations, on the other hand, can not be deducted from the class model and must be explicitly specified by the modeler. An example of interesting operation is any operation dealing with business logic of the application.

3.1.2 Describing relationships

Relationships in UML diagrams can have many properties and most of them must be modeled carefully. Very often modelers don't bother to specify properties of the relationships, accepting whatever default values the modeling tool provides. As we will see shortly, this can cause errors in the generated APIs. Of the main three relationship types the most interesting one is association relationship. Here we will look closely at it.

Navigability property of the association relationship deserves special attention. MDA API generators treat the navigable end of the association as a property of the opposite class. Thus, the navigable end must have a name and visibility, which are used to generate the corresponding property. This property must not be explicitly defined by the modeler in the class because this can cause redundancy. The class model should not define accessors and mutators that are already implied by the navigable association ends.

Association ends should be defined only when necessary because they enlarge APIs generated for the opposite class. Overuse of the navigability causes generators to produce unnecessarily large and complex APIs. Since generators treat a navigable association as a property of the opposite class, it is important not to give the same name to two ends of the opposite class. Failure to observe this rule leads to name clashes in generated APIs. Figure 3.2 on the next page illustrates this point. The *Order* class has two associations to the *Address* class. It means that an order contains two different addresses, which are *shopping* address and *billing* address.

An association end has a property named *aggregation* that can have one of the three values *none*, *shared*, and *composite*. This property must be also modeled carefully because MDA generators follow very specific rules about aggregation semantics and generate different APIs according to association's aggregation property. If the property is *none*, then the end is not an aggregate.

Composite aggregation is denoted with black diamond in UML. Figure 3.2 on the facing page shows an example composite aggregation. The semantics of *shipping* (and *billing*) address association is that existence of shipping address makes no sense without an order. If an order instance is destroyed then shipping address instance which is linked to it must also be destroyed. In the above example the multiplicity of the aggregation end is 1, which means that any *shipping address*

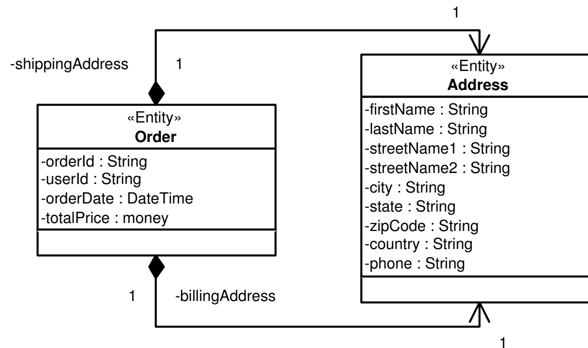


Figure 3.2: Navigable end name clashing

instance can be owned by only one instance of *order*. Multiplicity can also be 0..1 which in this case will mean that *shipping address* instance can exist without being linked to *order* instance. But when they become linked to each other then deletion of *order* will cause deletion of *shipping address* as well.

Shared aggregation is modeled with a white diamond in UML. It differs from composite aggregation in that an object O can be a part of several other object, not just one. Thus, the multiplicity of shared aggregation can be 0..* of 1..*. When one of the owners of O is destroyed, O may continue to exist as long as there exist other owners. However, when the last owner of O is destroyed, O must be also destroyed. Many modeling tools choose shared aggregation as a default aggregation type. Omitting to change aggregation type manually if necessary can cause semantic errors in the generated APIs.

Very often model elements are organized in packages. It is important to avoid mutual dependences between different packages. An example of such problem can be two classes residing in different packages and having navigable associations two each other. This couples both packages and makes it impossible to use any of them alone.

3.2 Pet Shop PIM

The Pet Shop Application is developed as collection of separate functional modules with well-defined interfaces. A *module* is not a formal UML concept. It is just an informal representation of a piece of software with specific behavior. In some ways modules resemble *UML packages* and *UML components*, but there are no direct correspondences between these concepts. Modular development structures software in several parts. This decoupling eases maintenance, simplifies parallel development, and provides opportunities for incorporating third-party components. The Pet Shop comprises following modules:

Control module — the control module dispatches requests to business logic, controls screen flow, coordinates component interactions, and activates user signon and registration.

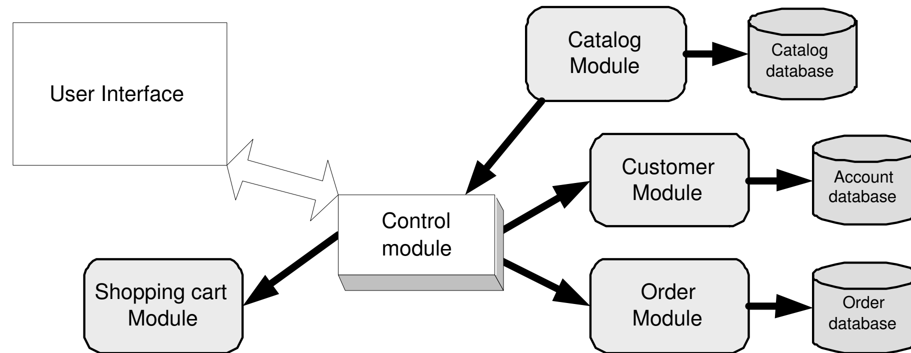


Figure 3.3: Pet Shop modules

Customer module — the customer module represents customer information such as addresses, credit cards, contact information, etc. In addition it is responsible for user authentication.

Catalog module — the catalog module provides a page-based view of the catalog based on user search criteria.

Shopping cart module — the shopping cart tracks the items a user has selected for purchase.

Order module — the order module sends order information to another application that manages order fulfillment.

Figure 3.3 shows how these modules interact with each other. It can be seen from the figure that the *Control module* plays the central role in the application. It serves the user requests by communicating with other modules. However, there is no direct interaction between the other modules. Such kind of design is very common in Web applications. The following subsections describe *Customer module*, *Catalog module*, and *Order module* in more details. We will look at the requirements for each module and their possible platform independent designs.

3.2.1 UML profile for Pet Shop Application

Before we proceed to describe PIMs of different Pet Shop modules we will look at a UML profile that I specifically designed for Pet Shop PIM. This profile identifies the main technology independent concepts that are common for all modules. These concepts reflect the business functionality issues and are not directly represented in the core UML. As we look at different Pet Shop modules, we will notice that we need two separate kinds of objects: process-oriented objects and data-oriented objects. The main difference between them is that process-oriented objects represent the flow control of the application and interaction with clients (other applications or other modules within the same application), while data-oriented objects represent persistent data.

The data-oriented objects have stereotype «Entity». Although all «Entity»

objects represent persistent data, there can be several differences in their persistence mechanism. In order to distinguish between different kinds of «Entity» objects, I associate two tagged values with «Entity» stereotype. The first one is called *DataSources*. It simply tells in what datasources the object's data can be made persistent. There are two restriction to that tagged value. The number of datasources for a single «Entity» object must be at least one and there must be no duplicates.

Another tagged value associated with «Entity» stereotype is called *canBeCached*. Its value can be *yes* or *no*. I defined this tagged value because it can influence application's design and it is independent of any particular platform. Suppose that we have a collection of «Entity» objects that represent countries. It is very unlikely that the total number of countries or their names will change several times a day. Therefore we can use some caching mechanism that holds collection of country objects in memory and refers to database only if necessary, thus sparing precious database connections. The concept of caching is well-known and frequently used both in .NET and J2EE platforms. On the other hand, there are many cases where caching is unnecessary. For example it would not help to hold a collection of catalog items in memory because it can be updated several times a minute. Objects that hold customer's data should neither be cached because they are used only for a short period of time and they are not shared among several clients.

When we have a collection of «Entity» objects we might be interested in finding one particular object (or subcollection of objects) that satisfies some criteria. The search criteria would typically be based on one of object's attributes. If the transformation algorithm knows what attributes can be used in search then it can automatically generate search methods. That is why I defined a stereotype «searchable» that can be applied to attributes of «Entity» objects. If an attribute doesn't have this stereotype then it can't be used as a search criterion. Sometimes we want to have an attribute that uniquely distinguishes all «Entity» objects in a collection. I used «unique» stereotype to identify such attributes. If an attribute has «unique» stereotype then it is automatically searchable and doesn't need to have an explicit «searchable» stereotype.

Process-oriented classes have «Controller» stereotype. The main properties of «Controller» classes are methods that perform some application specific logic. Sometimes methods of «Controller» classes require transactional behavior. A transactional method performs several operations that depend on each other's successful termination. An example of method that must support transactions is a method that makes several database updates which make sense only if all of them commit. I defined a special stereotype - «isTransactional» - which can be applied to methods of «Controller» class.

3.2.2 Customer module

The Pet Shop application supports two kinds of customers: registered customers and anonymous customers. Unregistered customers are allowed to visit *unprotected* pages (e.g. browse product catalog, put catalog items in a shopping cart). *Protected* pages (e.g. product purchase, updating personal information) are re-

served only for registered customers. Determining what pages are protected and what pages are unprotected is the one of the tasks of the Control module. In order to distinguish registered customers from unregistered customers the Pet Shop application must be able to store customer information in an external customer database and use that database for user authentication.

One of the decisions made when designing PIM of the Customer module was to separate it into two packages: *Customer SignOn* and *Customer Model*. The distinction between them will be discussed shortly. Figure 3.4 on the facing page shows how the PIM of the Customer module may look like.

The Customer module is responsible for managing information about customers of the Pet Shop. The main requirement of the Customer module is that customer data must be persistent and uniquely identifiable by the customer's system userId. The *Customer Model* package is designed to show what information is associated with a particular customer. This package doesn't contain any business logic. It's primary task is to define the structure of data that represents a customer and that can be exchanged between the Customer module and its clients (e.g. Control module). Figure 3.4 on the next page shows what information a customer account must contain. The model is intended to be self-explanatory. Here is a brief summary:

- A system-wide unique *userId* that corresponds to the userId that is used in authentication. *userId* can not be changed.

- Every user of the Pet Shop must supply a *password* in addition to userId to pass authentication. The password can be freely changed at any time by the user.

- Customer's *epost* is used by the Pet Shop application to notify the customer of certain events. An example is an order confirmation which is send to the customer after the customer submits a purchase order.

- *Profile* keeps track of the preferred language (the Pet Shop application can work with multiple languages), favorite product category, and determines whether a customer wants to see banners with product advertisements.

- *Credit card* contains necessary information about the credit card that customer will use to purchase products.

- *Address* contains name of the customer and common address fields such as street, city, zip code, etc.

The Customer module represents a collection of *Account* objects. It provides no direct access to any particular object. The reason to this is that there are some special rules about how these object should be accessed (for the first time) or created. Thus, we need a separate *Customer SignOn* package that would handle those issues. Figure 3.4 on the facing page shows that we have a *SignOnManager* with two business operations.

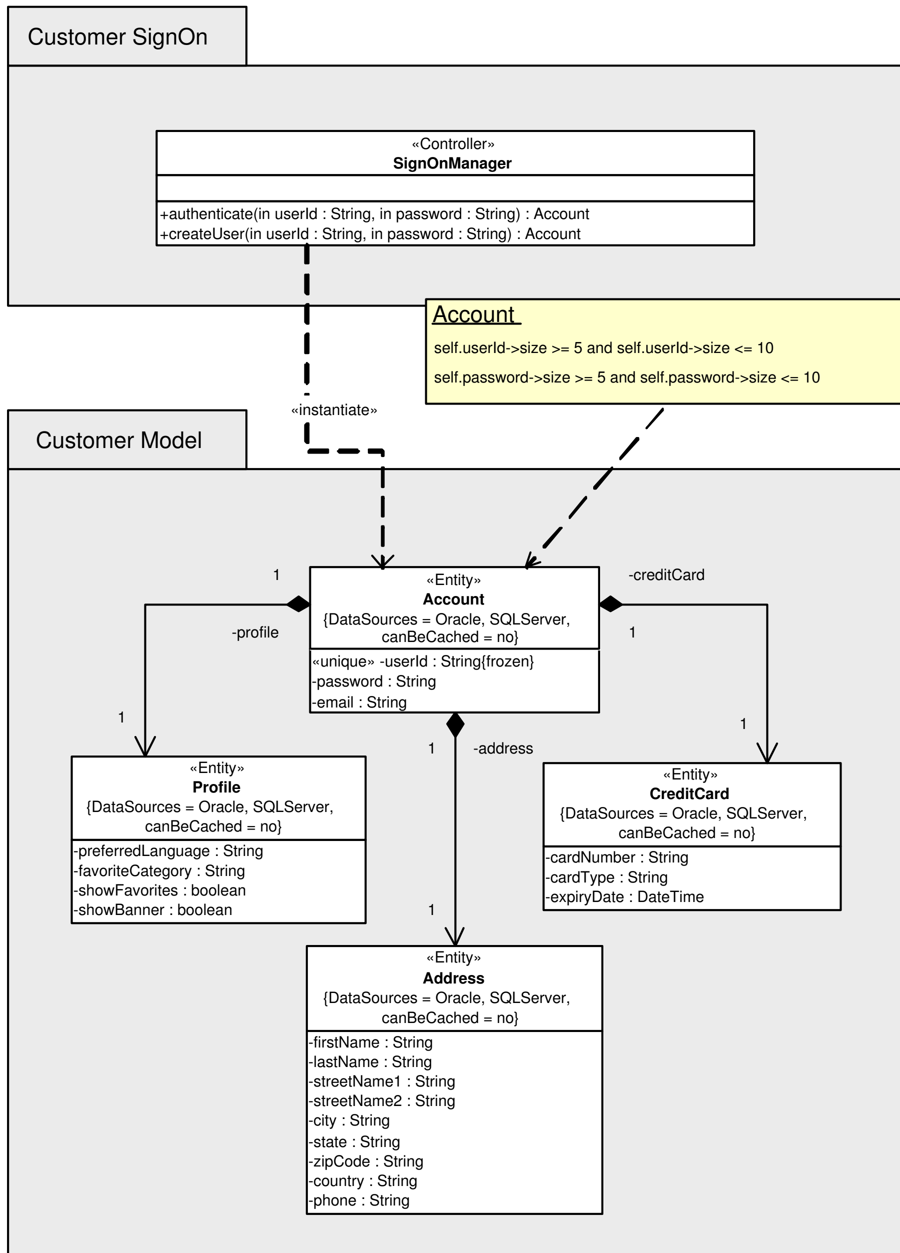


Figure 3.4: Customer module PIM

authenticate (sign on an existing user) — If the customer requests a signon as an existing user, the Pet Shop signs on the user, authenticating with the given userId and password. The return value of this operation is *Account* object containing customer information associated with the user.

createUser (sign on a new user) — If the customer requests creation of a new account, the Pet Shop creates a new account with the given name and password. This method should be the only way to create new accounts. Clients of the *Customer* module can not create accounts directly because of several important restrictions. Among them are the uniqueness of the userId which can not be checked by the client and specific constraints on userId and password (e.g. min and max number of characters, prohibited characters, etc).

3.2.3 Catalog module

The Catalog module provides an access to product catalog. The catalog contains multiple categories, which contain multiple products, which may contain multiple items. A client program accesses the catalog to retrieve information about catalog entries (categories, products, and items) either individually or page-by-page. A page may contain any number of either categories, products, or items, but all entries on the page must be of the same type. The Catalog module gets catalog entries from an external read-only data store. Here is the summary of the main requirements that the Catalog module must meet:

- Each individual item shipped by the Pet Shop enterprise must have a unique identifier plus descriptive information.
- A product is a group of catalog items, with an identifier, a name, and a description.
- A category is a group of related products, with an identifier, a name, and a description.
- The catalog module must also provide read-only access to list of categories, products, and items by "page". A page is a sublist of specific length, starting with a specific entry, of the fully-ordered list of categories, products, or items.
- The Catalog module must be usable by multiple client types.
- Because the catalog is the most highly-used component in the application, it must be as efficient and scalable as possible.
- The storage mechanism for the catalog contents should be easy to change.

Figure 3.5 on the next page shows how the Catalog module can be designed. We can see that the PIM of the Catalog module resembles the PIM of Customer module in a way that the functional and the informational parts of the model are divided between two packages. The *Catalog Model* package represents the internal structure of catalog including separate items and collections of them. *Catalog Product Finder* package provides the functionality to browse catalog and search for specific items.

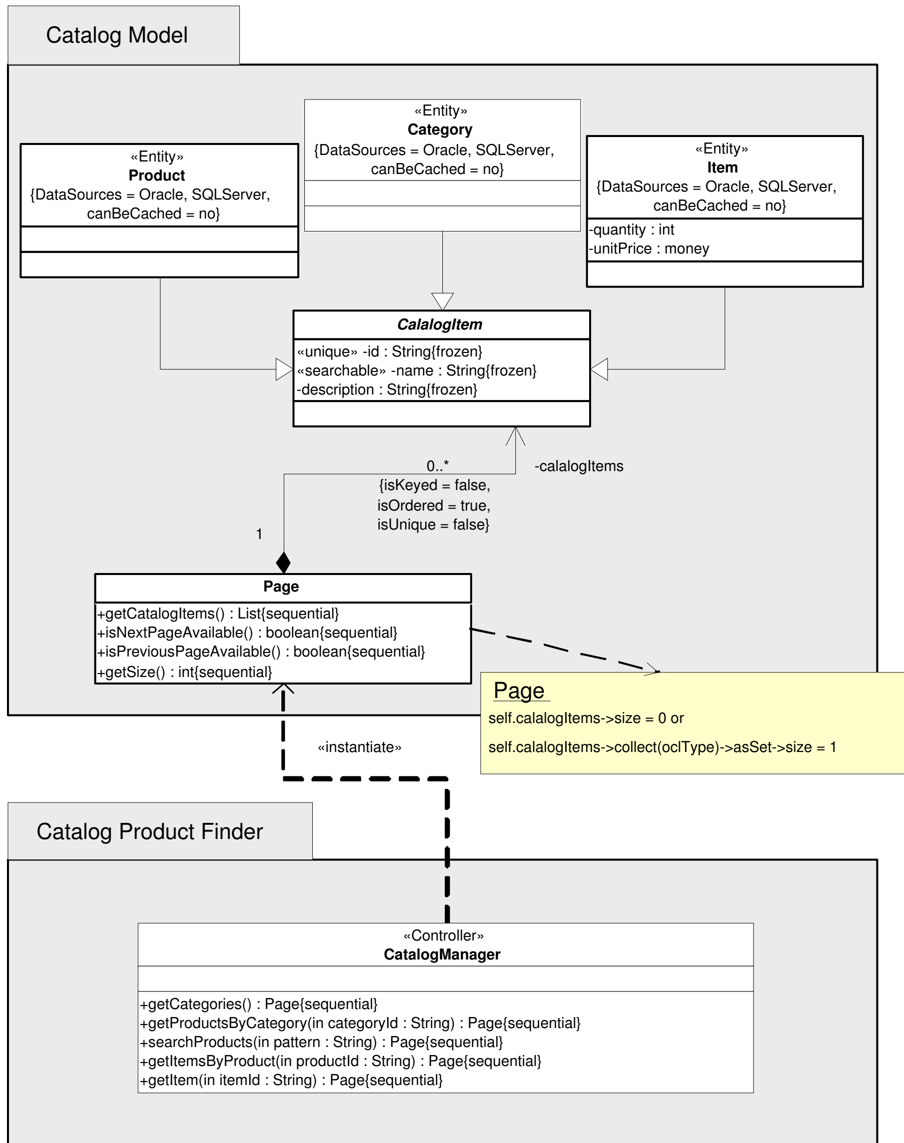


Figure 3.5: Catalog module PIM

CatalogManager controller class provides a number of ways to access product catalog. It is possible to get a list of all categories in a catalog (*getCategories*), list all products in a specific category (*getProductsByCategory*), list all items for a specific product (*getItemsByProduct*), search for products by providing some name pattern (*searchProducts*), and get a specific item (*getItem*). The *CatalogManager* provides no methods for modifying contents of the product catalog since all user access to product catalog should be read-only. The *CatalogManager* should be flexible in retrieving catalog data meaning that it must be able to work with arbitrary data storage.

All methods of the *CatalogManager* return a *Page* which is a representation of a page containing catalog entries. *Page* not only contains a collection of catalog entries but also has methods to determine whether there exist preceding or following pages (*isPreviousPageAvailable* and *isNextPageAvailable*). All catalog entries on a page are sorted by name. The OCL statement expresses additional requirement to the page containing catalog entries. It states that if the list of catalog entries is not empty then all entries must be of the same type.

The internal structure of catalog categories, products, and items are very similar. All of them contain id, name, and description. It could be possible to have only one class representing both categories, products, and items. However, these catalog entries are semantically different and it is a good practice to distinguish such elements in a model. I created an abstract class called *CatalogItem* that has three *frozen* attributes: unique *id*, *name*, and *description*. This class can not be directly instantiated but it serves as a base class for three concrete classes *Category*, *Product*, and *Item*. The *Item* class has two additional attributes: *quantity* (number of such items in the catalog) and *unitPrice* (price for one item).

3.2.4 Order module

The Order module is responsible for receiving a purchase order from a client and transmitting it to the Order Processing Center (OCL) for fulfillment. The communication between the Order module of Pet Shop and the OPC must meet the following requirements:

- Communication must be asynchronous. Because the order fulfillment process may take a long time, the Pet Shop must be able to create an order and continue, not waiting for the order to complete.
- Message delivery must be asynchronous. The OPC must receive and process the message exactly once.
- If the OPC is not available when the message is sent, the message must be stored and delivered when the OPC becomes available.
- Each order transmitted by the Pet Shop must include a globally-unique identifier.

The PIM of Order module is shown in figure 3.6 on page 48. It consists of two

packages: *Order Model* and *Order Processor*. The Order Model package represents information associated with a single order. The order consists *LineItems* of which correspond to catalog items, *shipping and billing addresses*, *credit card* information, *userId* of the customer that submitted the order, *date* of the order, and the total cost of all items in the order.

The *Order Processor* package contains classes with required business logic for order processing. The *OrderManager* class has two operations. *saveOrder* saves the order in the database and sends the order for fulfillment to the OPC. *getOrder* retrieves the order from the database. *IdGenerator* class is responsible for generating unique order id. Before the *OrderManager* can send the order to the OPC, it must ask *IdGenerator* for the id and assign it to the order. *saveOrder()* method must support transactions because it updates data from two different datasources.

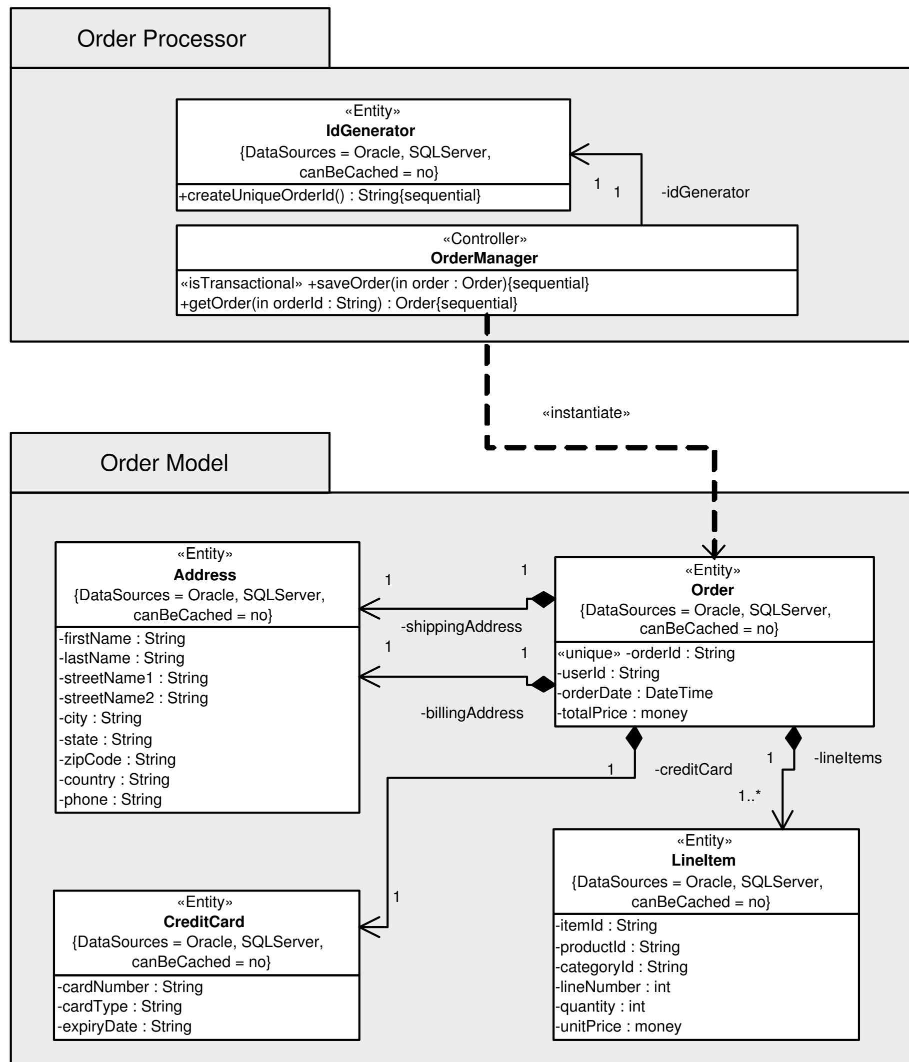


Figure 3.6: Order module PIM

Chapter 4

J2EE versus .NET platform

> J2EE technology and its component based model simplifies enterprise development and deployment. The J2EE platform manages the infrastructure and supports the Web services to enable development of secure, robust and interoperable business applications.
>
> *—from http://java.sun.com/j2ee/*

> Microsoft .NET is a set of Microsoft software technologies for connecting information, people, systems, and devices. It enables a high level of software integration through the use of Web services — small, discrete, building-block applications that connect to each other as well as to other, larger applications over the Internet.
>
> *—from http://www.microsoft.com/net/*

4.1 Introduction

This chapter gives an introduction to Java 2, Enterprise Edition and .NET platforms. All examples of MDA transformations in this document will be based on J2EE and .NET and it is important to understand how these two technologies relate to each other. The main two questions that we should be able to answer at the end of this chapter are:

- Are J2EE and .NET completely different technologies or they are just different approaches to solving identical problems?
- How big are architectural differences in .NET and J2EE?

At the beginning of this chapter I gave two quotations from the official cites of Sun and Microsoft. It is not hard to see that J2EE and .NET are advertised almost identically and there is a good reason for that. At first glance, J2EE and .NET are very similar. Both of them provide a reach set of services for building reliable, highly performing and scalable enterprise applications. This was in fact the main criterion for choosing J2EE and .NET as example platforms in

illustrating principles of MDA. As we saw in chapter 2 on page 18, PIMs can only be transformed to directly comparable platforms.

Going in depth, a comparison becomes almost impossible due to the completely different types of technologies involved. J2EE is a *standard* which is specified by Sun and implemented by a number of independent vendors. .NET, on the other hand, is a *product* developed by a single vendor. This difference gives a significant impact on J2EE and .NET architectures.

In this chapter I am going to compare J2EE and .NET platforms side by side. First, we will discuss the core part of each platform — the runtime environment. Next, we will look at programming languages that are used in J2EE and .NET. We don't have much choice with J2EE, since it uses only one language — Java. Some efforts have been made in adopting other languages, but was no success. The .NET platform supports a variety of programming languages (C#, VB.NET, C++, etc), all of which conform to some specific rules defined by .NET. Thus, the semantic difference between these languages are minor and my choice was the most used .NET language — C#.

When we are finished with the basics of both platforms we will compare enterprise application architectures of J2EE and .NET. This is perhaps the most important part to compare, because it is there the most of the functionality lies. I'm not going to tell which platform is better and which is worse. We will not discuss such properties of J2EE and .NET as performance, scalability, reliability or cost of development. Many different papers are written on this subject and this is of absolutely no interest in this discussion. My aim is to show where .NET and J2EE are similar and where they differ. This will help us to identify concepts that can be directly included in PIMs, concepts that must be modified before inclusion in PIMs, and finally, concepts that can not be included in PIMs at all.

Comparison of .NET and J2EE in this chapter covers only a small part of each platform. It is not intended to be a complete technical specification, but rather a brief introduction. For more information on J2EE and .NET consult [10], [9], [17], [14], [16], [19], [18] and other resources.

Introducing Java/J2EE...

J2EE is based on the Java programming language which appeared in 1995 and gained a great popularity among programmers worldwide. Java was originally designed as a programming platform for consumer electronic devices. The idea was to create a programming language that could run on different kinds of computers, consumer gadgets, and other devices. *Write Once Run Anywhere* advantage of Java made it a programming language of choice for many programmers because it allows to reuse once written code on many platforms. Another major advantage of Java is automatic garbage collection and absence of pointers.

Approximately three years after the initial release of the Java programming language, Sun released the first version of the Java 2 Platform, Enterprise Edition. The mission of J2EE was to provide a platform-independent, portable, multi-

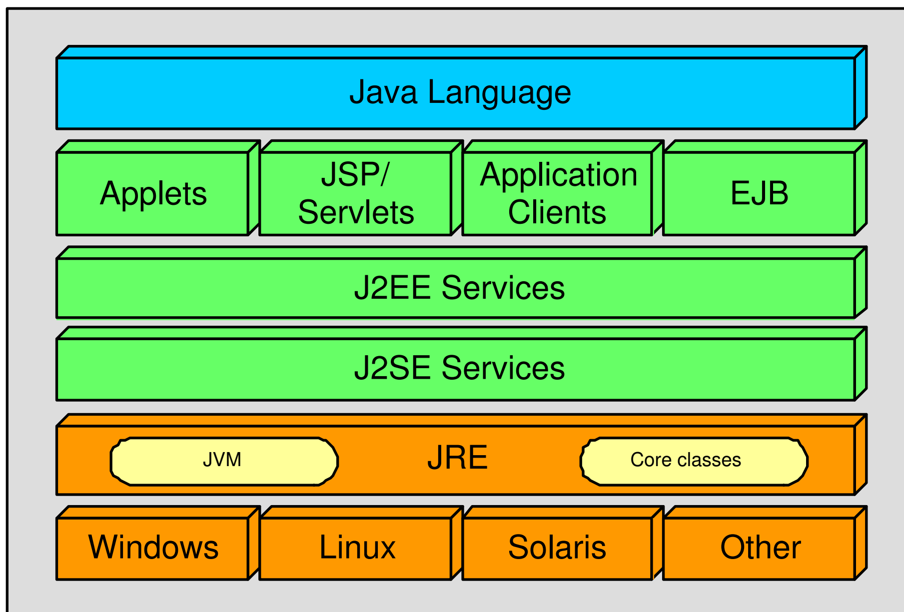


Figure 4.1: J2EE Platform overview

user, secure, and standard enterprise-class platform for server-side deployments written in the Java language. J2EE is an open standard and, thus, it promotes competition among vendor products and tools which leads to appearance of high quality middleware.

Introducing C#/.NET...

In June 2000, Microsoft announced the arrival of a new distributed software framework known as .NET. Microsoft .NET is a set of Microsoft software technologies for connecting the world of information, people, systems, and devices. While it is a general-purpose development platform, it has been designed from inception to use many open Internet standards and offers strong support for highly scalable distributed applications, in particular XML Web services. In other words, .NET is a platform for building integration architectures whose components are Web services that are interconnected through the Internet.

Together with .NET framework Microsoft introduced a new programming language called C#. C# was designed to take the maximum advantage of the .NET. It includes many features that made the Java language so popular, but adds some features that cannot be found in other programming languages.

Introducing architecture models

I conclude this section by presenting two figures (4.1 and 4.2 on the next page) which diagrammatically show the overall structure of .NET and J2EE platforms. These figures will be explained in the following sections.

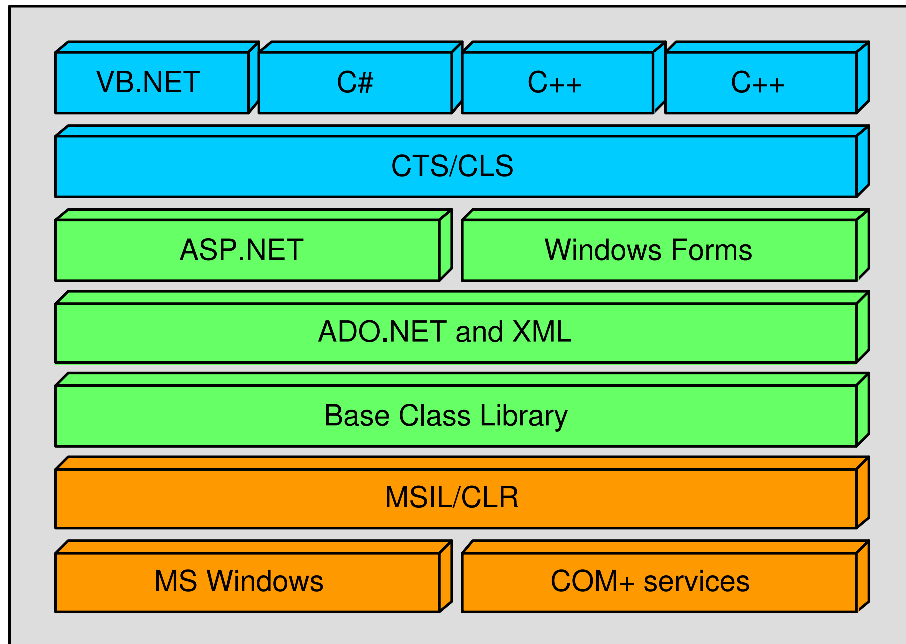


Figure 4.2: .NET Platform overview

4.2 Runtime environment

In this section I will compare runtime environments for J2EE and .NET platforms. The runtime environment is an infrastructure that is responsible for executing program code. Both J2EE and .NET make use of a managed space for running enterprise applications. It is called managed because it ensures that applications run safely and without unexpected behavior. However, the degree of such safety is different in J2EE and .NET. The managed space is responsible for garbage collection, error handling, type definitions, polymorphic method resolution, and other important features of any programming language. The runtime environment also includes a set of standard libraries that are frequently used in many applications.

I begin comparison of J2EE and .NET with comparison of runtime environments because this knowledge can be used in architecture modeling and thus, it can influence architecture design. For example, deployment diagram, which is usually a part of a MDA-compliant system architecture development methodology, can use such information.

J2EE

The core of the J2EE specification is the object-oriented programming language Java. One of its characteristics is the hybrid form between interpreter language and compiler language: the source code is translated into an intermediate, assembler-like format, the so-called Java byte code. This is a neutral

representation of the program which can be interpreted on every platform offering a Java Virtual Machine (JVM). Java byte code is stored in Java class files. Several such files can be packaged in JAR (Java archive) files.

The JVM itself depends on the platform used. JVM is a system process that reads Java byte code and executes it within a managed space. This allows running the same Java byte code on different platforms without recompilation, because different JVMs interpret Java byte code differently according to the rules and special features of the underlying platforms.

.NET

The foundation of the Microsoft .NET framework is Microsoft Intermediate Language (MSIL) and Common Language Runtime (CLR). The MSIL is the equivalent of the Java byte code system. When a source file containing any .NET-compatible language is compiled, the output consists of MSIL instructions. Like Java byte codes, MSIL is an intermediate representation and must be compiled to native instructions in order to be executed. .NET programs compile to something called an *assembly*, which is a standard exe or dll. Assemblies contain header information, MSIL and .NET metadata. MSIL code is sometimes referred to as *managed* code because the CLR manages its lifetime and execution.

The CLR, like Java Virtual Machine, is responsible for managing the execution of code and providing core services such as automatic memory management, threading, security, and integration with the underlying operating system. The difference from JVM is that MSIL can not be interpreted directly like Java byte code. Before MSIL can be executed, it must be translated into native code by CLR. This process is called *just-in-time*(JIT) compiling. .NET is optimized for compilation and is, thus, sometimes faster than Java.

Programming languages and Operating systems

Microsoft designed the .NET architecture to be used with several *.NET enabled programming languages* (any language that has MSIL interpreter). C# is the primary language for building .NET applications. However .NET applications can also be built using VisualBasic.NET, C++, J#, COBOL, and several other languages being adapted for .NET. Different parts of one application can be written in different languages.

.NET achieves language interoperability because MSIL was designed to accommodate the needs of any language. This is done with help of Common Type System (CTS). The CTS defines how types are declared, used, and managed at run time. The CTS is an important part of the .NET cross-language support and provides the basis for types written in one language to be used in another. Perhaps the most impressive example of cross language interoperability is the ability to define a base class in one language (say, C#) and override methods in a completely unrelated language (say, COBOL).

CTS will be discussed in the next section when we take a look at C#. For now

it is important to mention about the Common Language Specification (CLS), which is a subset of CTS. The CLS details for compiler vendors the minimum set of features that their compilers must support if these compilers are to target the CLR. Components that conform to the CLS are guaranteed to be usable by any other component that conforms to the specification. The CLR/CTS support a lot more features than the subset defined by CLS. The type that uses all features of a specific .NET enabled language could risk being not accessible from the types written in other .NET languages.

Microsoft .NET was specifically designed for Windows-based platforms such as Windows 2000 or Windows XP. It is theoretically possible to use .NET on other platforms and some projects on standardizing MSIL and CLR are currently underway. Some efforts are being made to port .NET to Linux and FreeBSD. The main problem is, however, that the most important part of .NET framework — enterprise services and components — are tightly integrated with Windows platform (discussed in section 4.4 on page 59). This means that much of the functionality of .NET can disappear on non-Windows platforms.

In comparison to .NET J2EE was designed to run on multiple platforms. The only requirement for executing Java programs on a specific platform is existence of the JVM for that platform. This is a big advantage of the J2EE platform. Unfortunately, this advantage can also be a limitation: it is often difficult or even impossible to take advantage of specialized hardware and operating systems.

Another difference of J2EE from .NET is that in J2EE the intermediate language was designed to meet the needs of Java. While it is possible to generate Java byte code from languages other than Java, in practice such projects are not considered to be mainstream J2EE development.

Summary

Table 4.1 summarizes the main ideas of this section.

Feature	**J2EE**	**.NET**
Managed execution environment	JVM	CLR
Intermediate language	Java byte code	MSIL
Time of compilation	Compiled as the program is executed	Compiled at deployment time or at load time
Programming languages	Java	C#, VB.NET, J#, ...
Operating systems	Windows XP, Solaris, OS X, ...	Windows-based OS

Table 4.1: Comparing Runtime Environment of J2EE and .NET

4.3 Java vs. C#

In this section I will compare the language systems of .NET and J2EE. The Java programming language is the only language of J2EE. In .NET, we have several possibilities. The most widely used .NET enabled languages are C# and

VB.NET. Many other languages can also be used in .NET but their application is very limited. I think that the most interesting language for comparing with Java is C# because:

- C# was specially designed for .NET to take the maximum advantage of .NET framework.
- C# is very similar Java.

C# is very similar to Java because both languages have almost the same core. C# was designed after Java and it took most of the best features that can be found in Java, though many of these features were given new names in C#. Since C# is a language for writing .NET applications, it defines many language constructs that support Common Type System. Most of these constructs are new to Java. We will compare C# and Java by looking at how they implement some of the main aspects of modern object-oriented languages.

C# and Java both derive from C and C++. Most significant features (e.g., garbage collection, hierarchical namespaces) are present in both. C# borrows some of the component concepts from JavaBeans (properties/attributes, events, etc.), adds some of its own (like metadata tags), but incorporates these features into the syntax differently.

Types

Both Java and C# have *primitive* and *reference* types. Primitive types are directly supported by the compiler. They include *byte*, *int*, *long*, and other. C# provides a richer collection of primitive types. Reference types include *class*, *interface*, and *array*. Objects of reference types are allocated on the heap. All primitive types in .NET are classes and therefore they can be used where objects are expected. In Java primitive types cannot be used as classes.

In Java, all types defined by a programmer are reference types. In C# this is not the case. C# defines a *struct* type (*value* type in CTS terminology) in addition to reference type. This kind of type has no analogue in Java. *Value* types are similar to *reference* types. The only difference is that *value* types are allocated on stack. Therefore, they contain data (not reference to data) and they don't need garbage collection. The process of converting value types to reference types is called *boxing*. The reverse process is called *unboxing*. Much of boxing/unboxing is done transparently.

C# provides *enum* types (short for enumeration). *Enum* is a data type that declares a set of named integer constants. *Enum* types have no direct equivalent in Java. The closest Java alternative is a set of individually defined constant values. Another C# type that doesn't exist in Java is pointer. Pointers are used in C# to access unmanaged code.

Structure of a Class

The CTS specifies that a type can contain zero or more members. Here is a brief introduction to these members:

Field A data variable that is part of the object's state. Fields are identified by their names and types.

Method A function that performs an operation on the object. Methods have names, signatures, and modifiers.

Property Properties provide means to manipulate object state but are not stateful mechanisms itselves. They look like fields to the caller. But to type implementer, they look like methods.

Event An event allows a notification mechanism between an object and other interested objects.

C# supports all these members. Java classes do not have the last two members. Java can easily simulate C# properties by using usual getter and setter methods. The situation with event is more difficult. C# introduces a *delegate* type that has no direct analogue in Java. Delegates provide an object-oriented type-safe mechanism for passing method references as parameters without using function pointers. Delegates are primarily used for event handling and asynchronous callbacks. *Action Listeners* in Java are, perhaps, the most appropriate constructs for achieving functionality that is similar to C# delegates.

C# provides the *indexer* member type. Indexers provide indirect access to some collection-related information in a class using array-style index. The index can be any value or reference type. Like properties, indexers can be implemented in Java with getter methods.

Automatic Garbage Collection

Both Java and C# automatically destroy objects that are no longer needed by an application during execution. One difference between the garbage collection mechanisms is that in C#, the programmer can force a garbage collection to occur. This can be done by using methods of the predefined .NET type *System.GC*. Although Java also has a command that would appear to do the same thing (*System.gc()*), the Java API documentation states that the garbage collection is not guaranteed to occur immediately after issuing that command.

Classes and Interfaces

Both Java and .NET support single inheritance within classes. A class in C# and Java cannot extend more than one base class. In Java, all classes extend a single root class *java.lang.Object* and in C# all classes extend a single root class *System.Object*. In both Java and C# if a class declaration does not specify a base class, the base class is assumed to be the root class. Classes in Java and C# can implement more than one interface. Both languages allow interfaces to extend multiple base interfaces.

Inheritance Modifiers

Table 4.2 on the facing page shows what inheritance modifiers can be applied to members in Java and C#. C# provides more inheritance modifiers than Java to accommodate the additional control over member inheritance in C#. The

Java	**C#**	**Effect on modified member**
abstract	abstract	*Abstract* classes can not be directly instantiated. All *abstract* members are virtual.
not available	new	*New* method should not override a virtual method defined by its base type.
default	override	Method overrides a virtual method defined by its base type.
final	sealed	Classes can not be subclassed. Methods can not be overridden.
default	virtual	Most-derived method is called even if object is cast to a base type.

Table 4.2: Inheritance modifiers of Java and C#

main difference between inheritance modifiers in .NET and Java is that in Java all methods are implicitly virtual. In .NET this is explicitly specified.

Constructors and Destructors

Both Java and C# support multiple constructors with different argument lists. Both languages allow for static initialization. Both languages allow methods (*destructors* in C# and *finalizers* in Java) to be called before an object is garbage collected. Neither destructors nor finalizers will always be called in every circumstance.

Namespace Partitioning

Names of classes and interfaces in Java are partitioned by *packages*. C# uses *namespaces* to partition names of classes and interfaces. Unlike Java packages, access boundaries are not enforced across namespaces. Also unlike Java packages, namespaces need not to correspond to any particular file system directory structure.

Access Modifiers

Both Java and C# allow different levels of access to be specified for methods and variables. Access modifiers (table 4.3) in both languages are quite similar. C# has one valid combination of access modifiers — *protected internal*.

Java	**C#**	**Accessible by**
public	public	anyone
protected	protected	subclasses
package (default)	internal	classes in the same package/assembly
private	private	members of the containing class

Table 4.3: Access modifiers of Java and C#

Main Method Entry Point

Both Java and C# use the *main* method as the program entry point. In addition to the *main* method that returns nothing, C# has a method that can return *int*.

Parameter Passing

In Java, all method parameters are passed by value. In C#, method parameters are passed by value unless prefixed with the *ref* or the *out* keyword.

When the *ref* keyword appears before a parameter in C#, the parameter is passed by reference and the reference must have been initialized prior to calling the method. When the *out* keyword appears before a parameter in C#, the parameter is passed by reference, and the method implementation must initialize the reference before returning from the call. C# also allows variable length parameter lists by using the *params* keyword on the last parameter in the parameter list.

Exception Handling

In C# all exceptions are unchecked. C# does not have a *throws* keyword. This means that the compiler does not notify programmers about exceptions that can occur. In Java, exceptions can be checked and unchecked. The Java compiler will generate compile time error if the programmer does not specify how checked exception should be dealt with.

Base class libraries

Any language (including Java and C#) need a runtime library it can depend on. Java 2 Standard Edition, Standard Development Kit (J2SE SDK) is the Java programming language's core API set. The core libraries in Java are the classes within the *java.** packages. C#'s runtime is the .NET framework's Base Class Library (BCL). The BCL is the basic runtime library which includes many classes in the *System* namespace.

If we look at the core libraries provided by J2SE and BCL, then we would see that these libraries are broadly comparable. Both of them contain utility functions, I/O functions, graphics, security, and much more. Table 4.4 on the facing page shows just few of them. We can see that the same functionality can be easily achieved in both Java and C#.

Summary

Comparison of Java and C# has showed that both languages are quite similar. However, there are some significant differences that must be taken into account when we try to make models that are independent of Java and C#. Here are the most important differences:

Class definition differences:

- Java doesn't have value types and enumeration types.

Classes	C#	Java
Basic Types	System.* (Type, Char, Int32, String, ...)	java.lang.* (Class, Character, Integer, String, ...)
Collections	System.Collections.* (CollectionsBase, Hashtable, ArrayList, Stack, SortedList, ...)	java.util.* (AbstractCollection, Hashtable, Vector, Stack, TreeSet, ...)
Calendar & globalization	System.Globalization.* (Calendar, RegionInfo, CultureInfo, DateTimeFormatInfo, ...), System.DateTime	java.util.* (Calendar, Locale, Currency, TimeZone, DateTime, ...), java.text.SimpleDateFormat
I/O	System.IO.* (Stream, StreamReader, StreamWriter, File, ...)	java.io.* (InputStream, OutputStream, InputStreamReader, OutputStreamWriter, File, ...)
Networking	System.Net.* (IPAddress, Sockets.TcpClient, WebClient, ...)	java.net.* (InetAddress, Socket, HttpURLConnection, ...)
Reflection	System.Reflection.* (MemeberInfo, MethodAttributes, FieldInfo, ...)	java.lang.reflect.* (AccessibleObject, Modifier, Field, ...)

Table 4.4: Example of base class libraries of Java and C#

- In C# all types are classes. Boxing/unboxing.
- Java doesn't have properties, events, and indexers.

Function signature differences:

- Slightly different access and inheritance modifiers in Java and C#.
- Difference in parameter passing.
- In C# all exceptions are unchecked.

Basic class library differences:

- Many classes with comparable functionality in Java and C# have different names and reside in different packages.

4.4 Comparison of enterprise application architectures

In this section I will continue to compare .NET and J2EE platforms by looking at their enterprise models. Both architectures allow enterprise-level, web-based applications to be developed and deployed. However, there are some important differences in the features provided by each architecture. I will try to identify these differences and divide them in two categories: completely incomparable features of .NET and J2EE and features that have different implementations in .NET and J2EE but still represent the same concepts.

In the introductory chapter 1 on page 6 we saw that a common approach to building enterprise applications is to partition them into three layers. The natural way to compare enterprise application architectures of .NET and J2EE is to look at each of these layers.

4.4.1 Presentation logic tier

Rich Client Layer

The primary means for building GUI clients in J2EE is Java Foundation Classes (otherwise known as Swing). Swing provides a rich set of graphical objects for building robust, complex user interfaces in a platform independent manner.

In .NET, GUI clients are typically implemented using Windows Forms. Developers writing client applications for Windows can use the Windows Forms application model to take advantage of all the rich user interface features of Windows, including existing ActiveX controls and new features of Windows 2000/XP, such as transparent, layered, and floating windows.

Java Applets are used to package Java code that will run within the browser. In the .NET platform, the same functionality can be achieved with ActiveX components. Applets and ActiveX components are not very important because neither of them are frequently used in enterprise applications.

Thin Client Layer

Thin clients are implemented in J2EE using servlets and Java Server Pages (JSP). Servlets can be used to programmatically generate any HTTP based content such as HTML documents. JSP can be used to embed Java code directly inside of HTML-styled documents called Java Server Pages. JSP also allows meaningful tags to be defined and used for generating dynamic content.

.NET uses Active Server Pages (ASP.NET) and WebForms to generate dynamic Web-based content. In ASP.NET, there are two basic ways in which a developer can separate the code from content:

- By using code-behind files (.NET classes that can be automatically generated from an Active Server Page and can control the page's behavior).
- By containing frequently used page logic in separate files called Pagelets (also known as page controls) and by reusing them in ASP.NET pages.

Both JSP and ASP.NET are translated and executed before delivering to client and from the client's point of view they are equivalent. However, there are two major differences:

- ASP.NET is tied to Microsoft Internet Information Server (IIS) while JSP/Servlets can run in different Web servers (e.g. Tomcat, WebSphere, etc.)
- ASP.NET can determine the ultimate destination browser and take advantage of that browser's functionality. With JSP/Servlets, it is the programmer's responsibility to figure out how the page should be rendered on the destination browser.

4.4.2 Domain logic tier

Component model

Both J2EE and .NET provide models for building software components. Components are reusable software elements that can be tied together to form software applications. In J2EE, the component model is the Enterprise JavaBeans (EJB) specification. The .NET platform equivalent to EJB is COM+.

There are very few architectural differences between EJB and COM+. Both are essentially derivative architectures from Microsoft Transaction Server (MTS), the original component-oriented middle tier infrastructure introduced by Microsoft in 1996. The key ideas first introduced by MTS and then incorporated into both EJB and COM+ included the following:

- High scalability through sharing of component instances.
- Middle-tier centric security.
- Automatic transaction boundary management.

Components allow applications to communicate across multi-tied client and server environments. Components come into play when there is a need for distributed objects to communicate which each other across either the client-server barrier or server-server barrier. The component model enables the application developer to work on the business aspects of the application without having to focus on transaction management, security or the life cycle management of the object itself.

Both J2EE and .NET component models have special environments in which components run (EJB calls them *containers*, COM+ calls them *contexts*). A component developer can specify how a particular component should behave by modifying specific attributes that define the component's behavior within a specific environment. For example, both EJB and COM+ allow to define the transaction behavior of a component by simply modifying configuration files without modifying the component's implementation. This feature is commonly known as "attribute based programming". COM+ also adds built-in support for component versioning, a feature that is not presently included with EJB.

There are three types of enterprise beans:

Entity Beans are *persistent objects* that can be stored in permanent storage. They use some persistence mechanism, such as serialization, O/R mapping to a relational database, or an object database. Entity beans contain core business data and don't perform any complex tasks. There are two ways to persist entity beans: *bean-managed persistence*(BMP) and *container-managed persistence*(CMP). With CMP, the container is responsible for ensuring persistence. With BMP, the programmer must implement persistence logic inside the entity bean.

Session Beans represent work being performed for client code that is calling it. Session beans are business process objects. They implement business

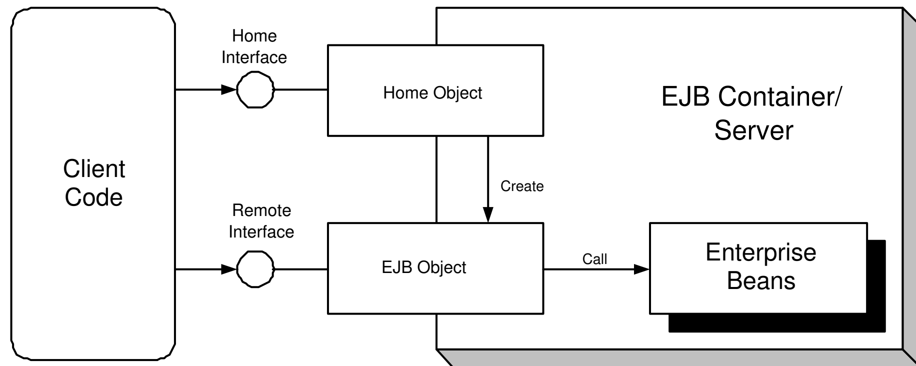


Figure 4.3: EJB model

logic, business rules, algorithms, and workflow. A session bean is relatively short-lived component. It has roughly the lifetime equivalent of a *session* or lifetime of the client code that is calling the session bean. There are two kinds of session beans: *stateful* session beans and *stateless* session beans. A stateful session bean is a bean that is designed to service business processes that span multiple method requests or transactions. A stateless session bean is a bean that that holds conversations that span a single method call.

Message-Driven Beans are special EJB components that can receive Java Messaging Service (JMS) messages. A message-driven bean consumes messages from queues or topics that are sent by any valid JMS client. A message-driven bean is decoupled from any clients that send messages to it. A client cannot access a message-driven bean through a component interface. It must use JMS API.

Figure 4.3 shows how enterprise beans work. An enterprise bean class contains implementation details for a component. Each enterprise bean class must implement a specific interface (*javax.ejb.SessionBean*, *javax.ejb.EntityBean*, or *javax.ejb.MessageDrivenBean*) depending on the kind of the bean.

When a client wants to use an instance of an enterprise bean class, the client never invokes the method directly on the actual bean instance. Rather, the invocation is intercepted by the *EJB Object* and then delegated to the bean instance. The EJB object is a network-aware intermediary between the client and the bean instance, handling necessary middleware issues. It is automatically generated by the container. To make this autogeneration possible the EJB object must implement *remote interface*(*javax.ejb.EJBObject*) that duplicates all the business logic methods that the corresponding bean class exposes.

To acquire a reference to an EJB object, the client code asks for an EJB object from an EJB object *factory*. This factory is responsible for instantiating, finding, and destroying EJB objects. The EJB specification calls such a factory a *home object*. Home objects must implement *home interface*(*javax.ejb.EJBHome*).

Home interfaces simply define methods for creating, destroying, and finding EJB objects. This is necessary because there can be several variants of each of these methods and container should be able to choose among them.

Local interfaces can be used instead of remote interfaces if all components reside on the same server and no networking is involved. In the case of networking, different communication technologies can be used. Usually, it is Sun's Java RMI-IIOP. Message-driven beans do not need home and remote interfaces at all. The reason is that message-driven beans process messages that come from messaging clients and they are never accessed directly.

.NET managed components can take advantage of automatic transaction processing, object pooling, and role-based security if they use COM+ Services. Components that use COM+ Services are called *serviced components*. The .NET class becomes a serviced component if derives from the *System.EnterpriseServices.ServicedComponent* class. This base class provides default implementation of the classic MTS/COM+ interface *IObjectControl* — Activate(), Deactivate(), and CanBePooled(). The default implementation can be overridden.

.NET objects that use COM+ services are automatically remotable because the required base class ultimately derives from *System.MarshalByRefObject*. This allows .NET Remoting infrastructure to marshal a reference to the object instance and enable serviced component to be remotely accessible from different contexts.

Table 4.5 gives some additional details about similarities and differences in .NET and J2EE component architectures.

Technology	**.NET**	**J2EE**
Business tier component architecture	COM+	EJB
Security API	ADSI	JAAS
Distributed transactions	MS-DTC	JTS
Message Queue API	MSMQ	JMS
Distribution protocol	.NET Remoting, SOAP	RMI/IIOP
Naming and Directory Service	ADSI	JNDI
Entity components	N/A	Entity beans
Built-in component versioning	Yes	No

Table 4.5: Component technologies in J2EE and .NET

Web Services Support

The next generation of distributed computing has arrived. Over the past few years, XML has enabled heterogeneous computing environments to share information over the World-Wide Web. It now offers a simplified means by which to share process as well. With web services, any application can be integrated so long as it is Internet-enabled. The foundation of web services is XML messaging

over standard web protocols such as HTTP. This is a very lightweight communication mechanism that any programming language, middleware, or platform can participate in, easing interoperability greatly.

The web services are performed using following industry accepted technologies:

- A provider creates, assembles, and deploys a web service using the programming language, middleware, and platform of the provider's own choice.
- The provider defines the web service in Web Services Description Language (WSDL). A WSDL document describes a web service to others.
- The provider registers the service in Universal Description, Discovery, and Integration (UDDI) registries. UDDI enables developers to publish web services and that enables their software to search for services offered by others.
- A prospective user finds the service by searching a UDDI registry.
- The user's application binds to the web service and invokes the service's operations using Simple Object Access Protocol (SOAP). SOAP offers an XML format for representing parameters and returns values over HTTP. It is the communications protocol that all web services use.
- ebXML is a suite of XML specifications and related processes and behavior designed to provide an e-infrastructure for B2B collaboration and integration where pure SOAP is not sufficient.

J2EE has recently been extended to include support for building XML-based web services. These web services can interoperate with other web services that may or may not have been written in the J2EE standard. J2EE implements basic web services technologies (SOAP, UDDI, WSDL, ebXML) through the Java APIs for XML (JAX API). In this way J2EE enterprise applications can publish own web services and connect to web services provided by other applications.

The .NET architecture was designed from the beginning with web services in mind. Therefore, .NET contains built-in classes and toolkits designed specifically for building SOAP-based services. Web services can be created in three different ways in .NET:

- Using MSXML, ASP.NET, or ISAPI interfaces (similar to JAX API).
- Using the built-in .NET SOAP message classes.
- Using the Microsoft SOAP Toolkit to expose functionality implemented by .NET managed components as web services.

4.4.3 Datasource logic tier

Database access

The primarily means for accessing relational data in J2EE is Java Database Connectivity (JDBC). JDBC is the Java-equivalent of ODBC, an API for accessing data in relational databases using SQL statements and cursor-based

access methods. The alternative to JDBC is SQLJ, which is a set of programming extensions that defines the interaction between the SQL database language and the Java programming language. SQLJ uses JDBC internally, but unlike JDBC, SQLJ permits compile-time checking of SQL syntax.

A complementary approach to JDBC is Java Data Objects (JDO). JDO is an architecture that provides a standard way to transparently persist plain Java objects. The programmer can write code in the Java programming language that transparently accesses the underlying data store, without using database-specific code. JDO hides SQL from the programmer. The main benefits of JDO are:

- A developer using the Java programming language does not need to learn SQL.
- Applications written with the JDO API are independent of the underlying database.
- Application programmers focus on their domain object model and leave the details of persistence (field-by-field storage of objects) to the JDO implementation.
- Application programmers delegate the details of persistence to the JDO implementation, which can optimize data access patterns for optimal performance.

EJB 2.0 CMP is the part of the J2EE component model that provides an object persistence service for EJB Containers. CMP's goal is to provide a standard mechanism for implementing persistent business components. CMP is not a general persistence facility for the Java Platform. CMP provides distributed, transactional, secure access to persistent data, with a guaranteed portable interface. CMP is based on a functional set/get data access model. It does not support transparent, Java instance variable persistence. EJB Query Language (EJB-SQL) is a standard and portable language for expressing CMP entity beans query operations.

Microsoft ADO.NET is the .NET equivalent of the JDBC API and provides the programmer with consistent access to a variety of data sources. Data sources can be relational databases or other tabular data sources, such as flat files or spreadsheets. ADO.NET consists of two major components:

The Data Provider includes the functionality required to manage a data source, including connection management, transaction support, and data retrieval and manipulation. The functionality of Data Provider is directly comparable with JDBC API.

The DataSet is a disconnected, in-memory cache of data. The DataSet provides a simplified relational data model in which data from multiple sources can be loaded an manipulated. Data can be written from a DataSet back to the original data source and forwarded to another component for further processing.

ADO.NET is highly integrated with .NET XML framework. The DataSet dynamically builds an XML schema inside to store the data from different data sources. Thus, ADO.NET can manage XML data in the same way as database data. Any data, regardless of how it is actually stored, can be manipulated as XML data or relational data depending on which model is most appropriate for the application at a given point in time.

As a conclusion to discussion of database support in J2EE and .NET, I summarize the main differences:

- JDBC is connection oriented, ADO.NET works offline.
- ADO.NET DataSet is an in-memory representation of database.
- ADO.NET is highly integrated with XML.

Integration with other enterprise systems

The J2EE Connector architecture defines a standard architecture for connecting the J2EE platform to heterogeneous enterprise application systems (EIS). Examples of EISs include enterprise resource planning (ERP), mainframe transaction processing, database systems, and legacy applications not written in the Java programming language. By defining a set of scalable, secure, and transactional mechanisms, the J2EE Connector architecture enables the integration of EISs with application servers and enterprise applications.

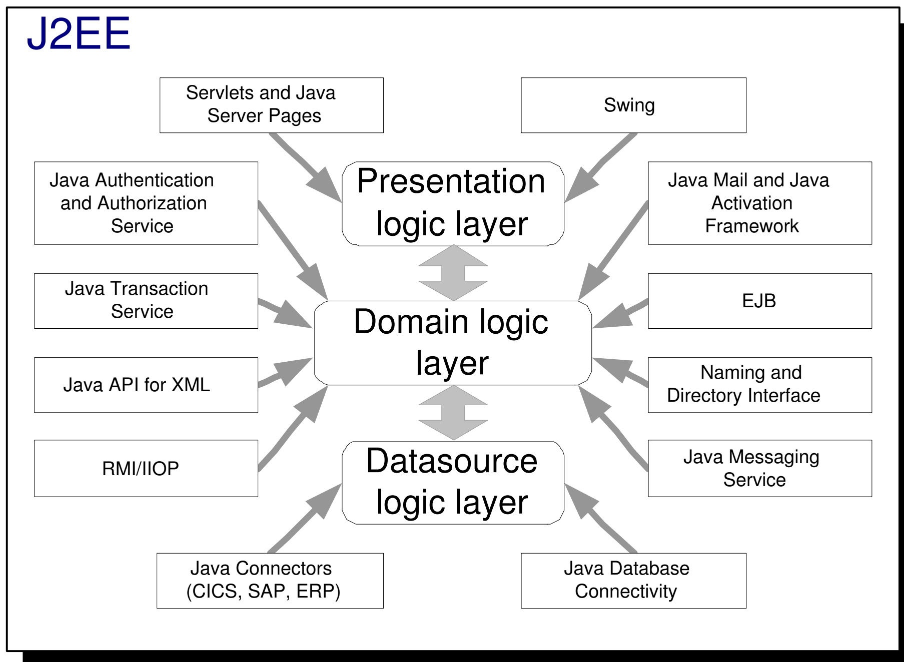
Similar to J2EE connectors, .NET provides the Microsoft Host Integration Server (HIS) connectivity to mainframes. HIS is not a core .NET component, but it may be employed as an extension.

Summary of J2EE and .NET enterprise application models

Figure 4.4 on the facing page shows how J2EE and .NET platforms provide necessary infrastructure for building enterprise applications. I showed only the most important components that are used at three main tiers of enterprise application architecture. In reality the number of components is bigger for both J2EE and .NET and many new components are now in development. The main idea of these figures is that all functionality provided by one platform is also provided by another platform. Implementation details of the comparable components may be different in .NET and J2EE but they still serve the same purpose.

4.5 Summary

In the beginning of this chapter I stated two questions regarding .NET and J2EE platforms. Now I can try to answer these questions. The first question was about similarity of these two platforms and the obvious answer is: both .NET and J2EE are equally suited for developing modern complex enterprise applications. Here I mention the main commonalities in J2EE and .NET:



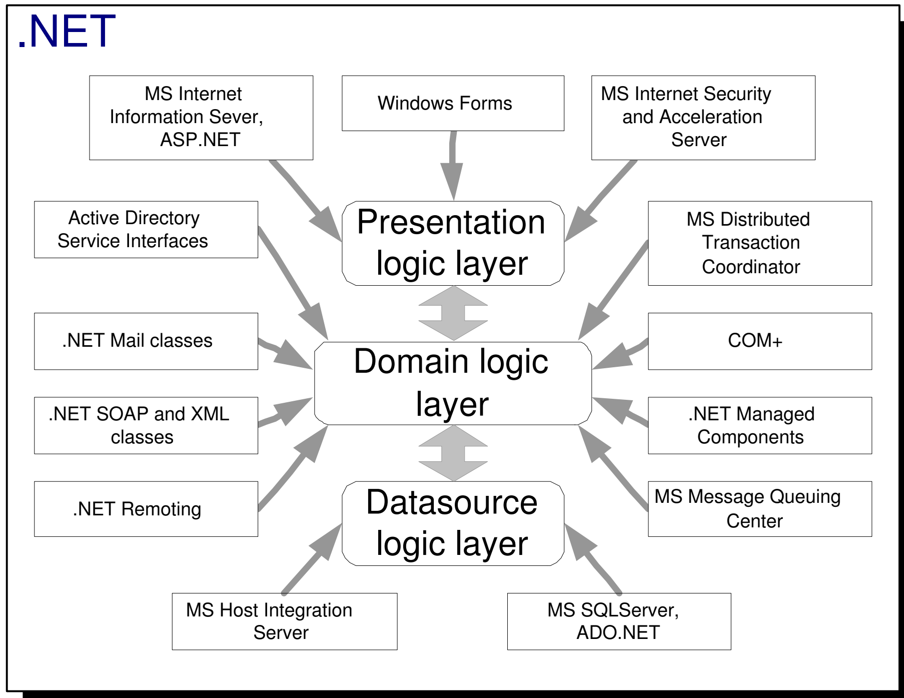


Figure 4.4: Summary of J2EE and .NET enterprise architectures

- Both J2EE and .NET are very popular platforms for enterprise application development.
- Both J2EE and .NET provide a rich set of services which meet almost all needs in current enterprise application development.
- Both J2EE and .NET are rapidly evolving to meet new challenges of enterprise application development.
- Both J2EE and .NET provide facilities to integrate with each other and even other technologies.
- Java and C# are very similar.
- Every enterprise application implemented in J2EE can be easily implemented in .NET and vice versa.

There are, of course, some differences. The first major difference is that J2EE is a standard and .NET is a product. All parts of the .NET platform are developed by Microsoft. In addition to core .NET framework Microsoft has many servers (e.g. IIS, HIS, BizTalk Server, SQL Server, ...) some of which are not part of the .NET framework but provide necessary services to it. In J2EE, almost all components/servers are provided by independent vendors.

Another difference is that J2EE is platform-independent and .NET is language-independent. Both forms of independence have certain benefits that can be important in some circumstances. J2EE is more mature technology than .NET. J2EE has been proven by industry for several years and many design choices have been optimized to give better performance.

The second question was about architectural differences of .NET and J2EE platforms. The answer to this question is following: while .NET and J2EE can seem equal from the end user view, there are many crucial architectural differences that developers must be aware of. Here are just few of them:

- .NET is more Web Services oriented and it is tightly integrated with XML. XML support is almost everywhere in .NET framework.
- There are many differences in language syntax of Java and C#. Some features of C# are not present in Java and vice versa. None of these features are of critical importance and many of them can be substituted with other design patterns. However, it can be a problem when we try to find related concepts in C# and Java.
- .NET doesn't have components with the functionality of J2EE's entity beans. Microsoft has indicated that an equivalent of entity beans will be included in future release of .NET framework. In the current version of .NET framework component persistence must be handled in some other way.
- ADO.NET takes a completely new approach to managing database data. Working with databases in a disconnected state gives an opportunity to

concentrate on business logic without thinking about expensive database connections. The possibility to handle data from different data sources in a uniform way and even to transform data between various formats adds much flexibility to datasource logic tier.

- ASP.NET has also some features that we can't find in JSP/Servlets. In ASP.NET all elements of a HTML page can be represented as usual objects. The main benefit of that approach is that every element of a HTML page can have a state associated with it. The state management of HTML elements is internally handled by ASP.NET infrastructure. In JSP/Servlets, the similar functionality can be achieved only with a significant amount of work from a programmer.

This is not a full list of all differences of J2EE and .NET platforms but it can give us a very important observation. *Although .NET and J2EE can do the same tasks, they do them in completely different ways.* This can present many problems when we try to combine PIM and PSMs.

It can be relatively easy to make a platform independent model which contains no J2EE or .NET specific details for an arbitrary enterprise application. The platform specific models for .NET and J2EE can be quite different but it would still be an easy task to make them. The problem arises when we try to combine all three models (one PIM and two PSMs) in one model.

It is not clear how we can put two different concepts under the same name in a common PSM. This means that we must be able to extract as many as possible comparable concepts from PSMs and find out how incomparable concepts can be showed in a common PSM without splitting them. This can be achieved by using some kinds of transformations that can transform a general concept to several unrelated concepts. It is the main topic of this thesis and in the next chapters we will look closely at different possibilities.

Chapter 5

Introduction to model transformations

MDA is about transforming models and model transformation is the central aspect of MDA. This chapter begins with a general discussion of the role of model transformations in MDA. After that we will look at the main properties of model transformations and existing techniques to implement model transformation tools. The main purpose of this chapter is to give the reader the basic understanding of the theory of model transformations not only in the aspect of MDA but also in the other fields of computer science.

5.1 Role of model transformations in MDA

MDA is about building models and performing different operations on these models. Models can be expressed in a variety of ways. In the previous chapters we saw at modeling with UML. Besides UML we can choose among many other representation technologies including XMI, JMI, CORBA IDL, and others. Even textual notation can be used to describe a model. Very often it is necessary to have several representations of the same model to allow model interchange and model analysis.

There exist different techniques to produce mappings from one notation to another. Sometimes these mappings preserve all model information, and sometimes they loose information which has no equivalent in an alternative modeling syntax. However, it is important to understand that such mappings are not real model transformations — they are merely alternative representations of the same model.

A model transformation occurs when models are refined and details are added for the purpose of focusing on a particular implementation technology or an aspect of the domain model. Model transformations are used to document different *levels of abstraction*, *viewpoints*, or *aspects* of an information system. Figure 5.1 on the next page shows the main four abstraction levels in MDA architecture. A typical MDA process can be seen as a chain of several transformations beginning at the most abstract model of the system (sometimes even informal) and ending

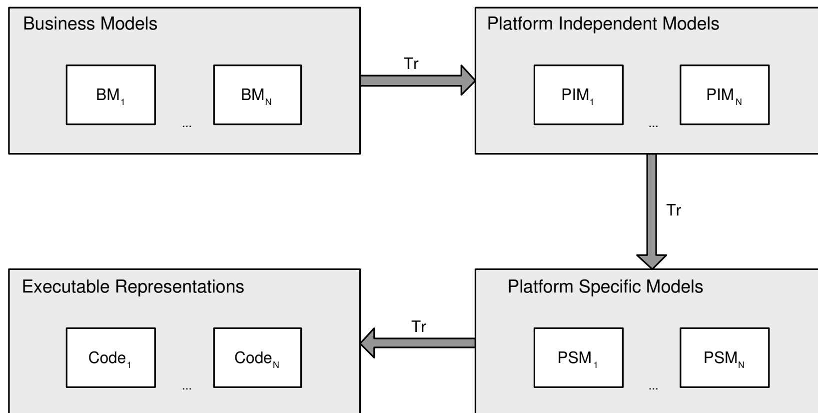


Figure 5.1: Use of model transformations in MDA

at concrete representation of the system (executable code).

The business (or domain) models are the view of the business person. Typically, domain models document the business from a logical perspective. Business models often lack details necessary for good software design, however resulting IT models must be consistent with the business model. Though I dropped designing business model of the Pet Shop in this paper, business models are considered to be an important part of MDA framework.

Business models are transformed into Platform Independent Models. The role of PIMs is covered in more details in the previous chapters. There can be several levels of PIMs beginning with the model which is totally independent of any technology. Subsequent levels of PIMs can add some technology issues. They conform to a particular technology paradigm (such as component technology, distributed objects, asynchronous messaging, etc). Obviously, additional transformations are required between different levels of PIMs. There can be several PIMs at the same level if they can not be ordered by abstraction.

A Platform Specific Model is the realization of a PIM in the definition syntax of a particular technology platform. For example, we can be interested in two UML models (PSMs) which focus on different platforms (e.g. J2EE and .NET) for the same PIM of the enterprise application. In this paper we will discuss only transformations of PIMs into PSMs. This kind of transformation is one of the most important parts of MDA (which is integrating applications built for different platforms). The main principles of PIM-to-PSM transformations can be freely used in all other kinds of model transformations.

Ultimately, the model must be realized in software. The execution code is also a model and the last transformation represents a code generation. The extent to which the PSM supports application logic will determine the extent to which software can be generated. The language that supports the PSM typically falls short of the full capabilities of a programming language. This kind of

transformation can reuse some techniques from PIM-to-PSM transformation but generally the code generation is a more complex process with some special needs.

Code generation from a PSM is not necessarily a MDA transformation. If the source PSM is so detailed that it is just a diagrammatic representation of the code structure then the code generation from PSM would not change level of abstraction. PSM-to-code transformation can be considered to be a MDA transformation only if executable code adds some additional semantics that is missing in the PSM.

There are two comments to the above discussion. The first comment is related to the concept of *model synchronization*. The figure 5.1 on the preceding page supposes that all manual changes are allowed only on the model that lies at the top of model hierarchy. This is implied by the fact that all changes are propagated only downwards. If we, for example, add any changes to PSM, we can risk that these changes will be overwritten (and lost) later when applying the original PIM-to-PSM transformation. This is because the PIM is totally unaware of any changes.

To cope with this problem we should allow transformations to be bidirectional. When we make some changes to any model all other models must be updated to reveal these changes. This is only possible if we can perform reverse transformations (Code-to-PSM, PSM-to-PIM, and PIM-to-BM) in addition to the transformations shown in figure 5.1 on the page before. The ability to transform models in any direction is referred to as model synchronization. We will not discuss reverse transformations any more but we should be aware that making transformations bidirectional can be a problem. A transformation rule can be seen as a function: given some input in the source model, it produces a concrete result in the target model. However, since different inputs may lead to the same output, the inverse of the transformation rule may not be a function because it will return a multiple (could be infinite) number of possible solutions. Another problem is that inversing a set of transformation rules may lead to fail to produce any result due to non-termination.

Another comment to the above figure is the question about necessity of explicit PSMs. Many researchers suggest that PSMs can be removed from the transformation chain. The main argument for that proposal is that there can be very many different models at different abstraction levels and it can be a difficult task to maintain them all. Of course, PSMs can not be removed totally because they contain valuable platform specific information. Without this information no code generation will be possible.

If we compare PIMs and PSMs we can observe that they are quite similar. The PSM represents the corresponding PIM plus platform specific information. This means that by keeping both PIM and all derived PSMs we in some way duplicate a lot of details in several models. The idea is to extract platform specific information from PSMs and use it later with PIM at code generation phase. This information should not become a part of PIM, it should be kept as a separate module. It will also be possible to construct PSMs (e.g. for better

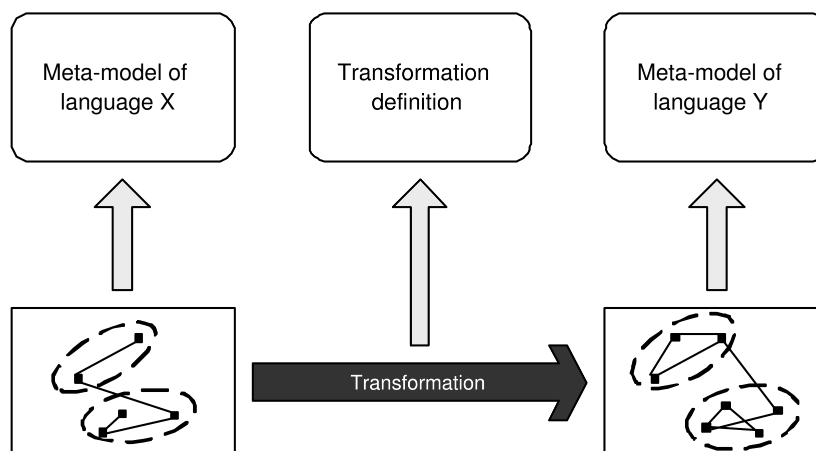


Figure 5.2: General prinsiples of model transformations

understanding and analyzing of the system), but PSMs will not be a necessary part of MDA process. We will take a closer look at this problem later in the next chapter.

5.2 Basic concepts of model transformations

Figure 5.2 shows a general scenario for model transformations in MDA. It shows that a transformation can be performed between two arbitrary models expressed in different languages and describing the system at different abstraction levels. However, there are two requirements that must be satisfied to allow such transformation. The languages of source model (X) and target model (Y) must be formally defined by metamodels. The transformation algorithm must also be formally defined. Ideally, it will be defined with a metamodel of transformation language.

Although source and target models can be defined in different languages, the most usual case would be both models expressed in the same language (e.g. UML) as in PIM-to-PSM transformation. The target model should also be at a lower abstraction level then the source model. The original model and the transformation rules can be placed in the MOF repository. But the MOF repository would typically have no knowledge about how the transformation should be performed. Additional code is required to execute transformation rules. There are several approaches to generating transformation code. The first one is to create a generator that reads the transformation rules and generates the transformation code that executes rules on the original model. Another implementation strategy is to write a generic transformer that reads transformation rules dynamically at runtime and executes the transformation of the original model.

Model transformation is closely related to model *mapping*. Mapping of models is about identifying associations between different models. The process of model

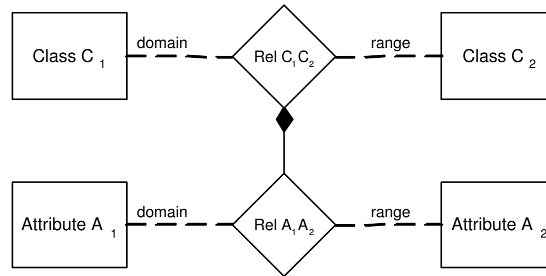


Figure 5.3: Example of relation syntax

mapping involves specifying how a particular element from one model is represented in the corresponding model after completed transformation. Elements in the original model are refined into (usually) more complex elements in a target model. A refinement is a complete transformation that preserves meaning and derives more complex patterns than it matches. The process of model transformation consists of searching for the known patterns (pattern matching) in the source model and applying transformation rules on these patterns (derive new patterns). Figure 5.2 on the page before shows correspondences between different model elements(in ovals) before and after transformation.

[6] distinguishes two different concepts in model mappings: *relations* and *pairs*. A pair contains two elements stemming from different models that are related to each other. A relation is a set of pairs. It constraints the way the pairs can be constructed by specifying sets of possibly connected model elements. To describe a mapping between two models, one defines a series of relations between elements from both sides. Both language definitions are separated in their own packages and by using relations their connection can be expressed without influencing the contents of those packages. Mechanisms are necessary to restrict the sets of objects on which the relation is applicable. These sets are called the *domain* and the *range* of the relation. The domain and range definition may comprise single elements or element tuples.

Relations are not independent of each other. Rather, connecting structures between the relations reflect the structure of the connected metamodels. An example of this is the mapping of nested structures. Figure 5.3 uses a UML-like graphical notation to show a mapping between different model elements. This figure expresses that class C_1 is related by relation C_1C_2 to class C_2. In this example both domain and range of the relation C_1C_2 are composed of single classes. There can be more complex relations where domain and/or range of the relation consist of several model elements. Mapping between two classes will result in the mapping of composed objects (class attributes). Thus, the attribute mapping can exist only in the *scope* of the class mapping. Other associations between relations (e.g. references to other mappings) are possible. These enable modular and non-redundant mapping specifications which are better maintainable.

A relation may specify additional constraints on the pair it contains. An example of such constraint may be preserving class names (but not class structure) during transforming class elements. If these constraints are precise and complete it is possible to *compute* a relation, i.e. given one element of a pair we can construct the corresponding element unambiguously. If a relation is said to be fully computable, then this works for both directions (domain to range and range to domain), yet most relations in practice will allow this kind of construction only for one direction. Furthermore, a relation can contain specification on the set of its tuples, requiring the set to be *complete* in regard to the domain and range. Clear distinction between terms *pair* and *relation* is not always made in model transformations and the more general term *mapping* is used instead.

We can define several requirements to model mappings and model transformation rules that are desirable if we want to benefit from model transformations. Among them are:

- Mappings should represent the implicit connection between their parts. They should neither be biased toward a particular direction not toward a specific application scenario. Mappings that are thus *independent* of their application will allow for transformations in both directions and the reconciliations of incremental changes.

- Model transformations rules should *depend only on metamodels* of target and source models. In this case the same transformation rules can be applied to any source model that is based on the specified metamodel.

- Model transformation rules must represent *generic tasks*, not depending on the level of abstraction. Such generic rules can be used at different phases of the MDA process.

- A *notation* of mappings should be *understandable* to enable their definition and reception by human users. The notation should allow for clear and uncluttered presentation of the relevant aspects, yet it should be precise and powerful enough to express properties of the mapping. A formal definition of the mapping notation is necessary to enable automatic processing and manipulation of mappings.

- There should be notation both for defining the model, and for viewing *examples* of the model. Amongst other things, the latter allows particular instances of a mapping to be examined. This could be important when particular instances of mapping need to be manipulated and/or corrected manually.

- Mappings should *not interfere* with the definitions of the models on either side, thus restricting the use of general purpose tools and theories. Rather, a clear separation of both models and their connection is desirable.

- Mappings should have *persistent instances*. If a mapping is completely computable, these can serve cached results from that computation. It the mapping is not computable they document the user's decisions in that matter. These persistent instances allow the tracing of origins and the reconciliation of incremental changes.

5.3 Overview of the existing approaches to model transformations

The concept of model mappings and model transformations is a fundamental concept of computer science. In the area of Software Engineering it appears in numerous approaches and application fields. Model mappings have been used for many years in:

- *Code generation* approaches that transform a UML model into program code.
- *Reverse engineering* approaches that allow to recover abstract model information from a given code.
- *Data and application integration* scenarios where mappings allow to denote related data and to propagate incremental changes.
- Integrating different data models (ER, relational, object-oriented, object-relational, etc)
- Expressing relations between syntax and semantics in *language definition*.
- Managing and tracking the connection between models at different stages of a *system development*.
- And many more...

We can see that we possess some experience in defining model transformation rules and implementing model transformation tools. In this section we will look at the commonalities and the differences of the existing approaches to model transformations.

5.3.1 Main features of model transformation approaches

In this section we give a brief summary of the main aspects of model transformations and the ways different model transformers address these aspects. More information about these aspects can be found in [4]. We can see that though approaches to model transformations can vary significantly, they all try to solve the same problems.

Transformation rules A transformation rule consists of two parts: a left-hand side (LHS) and the right-hand side (RHS). Both LHS and RHS can be represented using any mixture of variables (model elements), patterns (model fragments), and logic (computations and constraints on model elements).

Rule application scoping Source scope is the scope of the source model that is considered for rule application. Target scope is the scope of the target model, in which the RHS will be expanded.

Source-target relationship Some approaches mandate the creation of new target model that has to be separate from the source model. In some other approaches, source and target is always the same model. Yet other approaches allow setting the target scope to a new model or (possibly parts of) an existing model, which could be the original source model.

Rule application strategy A rule needs to be applied to a specific location within its source scope. Since there may be more than one match for a rule within a given source scope, we need an application strategy. This strategy could be deterministic or non-deterministic.

Rule scheduling Scheduling mechanisms determine the order in which the rules are applied. The scheduling mechanism can be implicit (some rules depend on the results produced by other rules) or explicit (the transformation rules are completely separated from the scheduling logic).

Rule organization The rule organization issue is concerned about expressing and packaging transformation rules, separating transformation rules from source and target models, and reuse of transformation rules.

Tracing Transformations may record links between their source and target elements. These links can be useful in performing impact analysis, synchronization between model, model-based debugging, and determining the target of transformation.

Directionality Most rules are applied in one direction by binding the LHS in the source and expanding the RHS in the target model. In some cases a declarative transformation rule can be applied in the reverse direction, too. An alternative approach is to define two separate rules, one for each direction.

5.3.2 Overview of model transformation approaches.

Researches had been investigating model transformations long before the OMG started the MDA initiative. Some looked into refactoring of models. Others tried to transform UML models to more formal models such as Petri nets to apply verification techniques. Furthermore, the aspect-oriented-programming community developed ideas on how to perform aspect weaving on the model level. Despite of the variety of the existing approaches to model transformations, all of them try to perform the similar tasks. After all, the process of transforming a model can be seen as a sequence of four steps which can be repeated many times:

- Searching for a certain pattern in the source model.
- Determining relevant elements of the target model under construction — that is, a pattern search in the target model.
- Adding, modifying, or removing elements in the target model — that is, applying a certain pattern.
- Bookkeeping of the already generated elements in the target model.

All existing approaches to define model transformation rules can be classified into three major categories: *model transformations using programming languages*, *model transformations that build on XML*, and *rule-based transformations that rely on expressing transformation rules in visual notation.*

Transformations using programming languages

This approach builds on traditional programming languages to implement the model transformations. Here we can distinguish three different kinds of languages: general purpose imperative languages (e.g. Java, C++), declarative languages (e.g. prolog, OCL derivatives), and database languages. All of them have special areas of application.

Solutions based on database languages allow mappings between different database schemas. Such mappings are very important because the variety of database models leads to the problems of interoperability, schema evolution, data integration, database optimization, etc. To satisfy these needs, several languages were proposed (e.g. SchemaSQL for relational databases and MetaOOL for object-oriented databases) that extend query languages by adding new notations. Some languages can easily express a mapping between different database models (e.g. RelOO can map between relational databases and object-oriented databases). Other languages such as HiLog or SchemaLog are based on new formalisms (mathematical notation, logical notation, etc).

OMG has defined a CWM (Common Warehouse Metamodel) which serves as a foundation for describing different database models in a uniform way. This standard can be used for integration of data from different data sources. The CWM transformation framework provides a mechanism for linking source and target elements, but the derivation of the target elements has to be implemented in some concrete language, which is not prescribed by the OMG. All mentioned database approaches have one big limitation: they are data-oriented and are not suited for arbitrary model transformations. However, many ideas can be used in other types of transformations.

General purpose programming languages are mainly used for transformation of UML models. This kind of model transformation is based on the fact that UML graphical notation is only one of many possible ways to express a MOF-based model. Some commercial and some open source UML tools provide access to their models via a general programming language with a specific API. Thereafter, programmers can use these APIs to create, manipulate and modify models. This approach is very attractive since the programmer manipulates the model directly in terms of the source and target metamodels. All the programmer needs to know is the model access API. In addition it can use all facilities provided by a general-purpose language (such as Java) to achieve transformation. However, it has a major drawback: the rules are embedded in the code. It is not easy to alter the transformation algorithm and understand the overall strategy of the transformation.

Declarative languages are widely used to describe relations, often using mathematical formalism. This approach is commonly referred to as relational approach. Its basic idea is to state the source and the target element type of a relation and specify it using constraints. In its pure form such specifications are non-executable. However, declarative constraints can be given executable semantic, much like in the case of logical programming. In fact, logic programming with its unification based matching, search, and backtracking seems a natural choice

to implement the relational approach, where predicates can be used to describe the relations. A big advantage of relational approach with logic-based programming language is the support for bidirectionality. Some work has also been done on expressing transformation algorithms with functional languages([7]), Object Constraint Language derivatives([1]).

XML based model transformations

XML is another way to represent MOF-based models. The primary purpose of encoding models as XML documents is to allow a standardized interchange (XMI) of models between different MOF tools. The XMI specification proposed by OMG is strongly coupled with MOF specification because it allows saving data and metadata that are MOF compliant in XML. The XML specification comes with a lot of standards. One of these is called eXtensible Stylesheet Language Transformation (XSLT) and is dedicated to the transformation of XML document into another document (XML or not) [20]. This language uses XPath to address XML document parts, which should be transformed.

It seems straightforward to transform a model by applying XSLT scripts on the exported XMI document and to re-import the transformed model [12]. However, this approach has several disadvantages. I can mention just a few of them:

- There exists a large semantic gap between a model expressed in UML notation and the corresponding serialized model (XMI). It means that it can be difficult for human users to analyze XML-encoded models.
- XSLT is designed for transforming tree-based data structures, where as a model is an arbitrary shaped graph.
- Writing an XSLT program is long and painful. A model transformation code can take several thousands lines. An important problem with XSLT is its poor readability and the high cost of maintenance for associated programs.
- Executing XSLT program is not user-friendly for model transformations. The error messages produced XSLT processor are not dedicated to model transformations. Most of the important transformation errors (non-existent concepts, bad relations, etc) will not be considered as errors by an XSLT processor.

Rule-based transformations based on visual notation

Rule-based transformations rely on describing model mappings using visual notations. The main advantage of this approach is that it is easy to understand and analyze transformations because the model mapping rules are separated from the program that performs transformations. In this scenario the generic transformation program can take the source model and mapping rules as input parameters and return the transformed model. The biggest problem is how to express mapping rules in such a way that they can be accepted by transformers. There are three main strategies to define transformation rules in a visual notation. Among them are: *textual notation*, *graph-transformation-based approaches*, and MOF-based approaches.

The most usual form for the definition of mappings between models is the use of natural language. This tends to be a rather vague form of definition, often relying on some examples to explain the meaning. While this might be intuitive to humans, it completely lacks the formal definition needed for automatic processing of the mapping rules. Completeness and unambiguity are hard to reach with natural language. That is why this approach has little practical value for our purposes.

There have been some proposals to define mapping rules partially by using natural language and partially by using a correspondence table, which associates concepts belonging to the target model with the concepts of the source model. This approach in some other specifications can be completed by defining constraints on mapping rules. So, if a formal language such as OCL expresses these constraints, then it is possible to verify that the process of transformation is correctly executed. This kind of specifying mapping rules is suitable for simple translations such as one-to-one correspondences, but it is unsuitable for a majority of transformation processes which require some intermediary calculus or more complex transformations than one-to-one mapping.

Graph-based-transformation approaches draw on the theoretical work on graph transformations. In particular, these approaches operate on typed, attributed, labeled graphs, which is a kind of graphs specifically designed to represent UML-like models. Graph transformation rules consist of a LHS graph pattern and a RHS graph pattern. The LHS pattern is matched in the model being transformed and replaced by the RHS pattern in place. Some additional logic is needed in order to compute target attribute values (such as element names). The graph-transformation-based approaches are powerful and declarative, but also the most complex ones. The complexity stems from the nondeterminism in scheduling and application strategy, which requires careful consideration of termination of the transformation process and the rule application ordering. There exists some experience with research prototypes.

OMG suggests to define mappings between models in a formal way using diagrammatic visual notation to express mapping rules. This approach seems to be very convenient because the language of mapping rules can be formally defined by a special metamodel that is similar to metamodels of the target and the source models. Furthermore, the mapping rules can use the standard graphical notation (UML) that is familiar to developers. Thus, the mapping rules can be defined in a separate model which contains elements from both models (source and target) connected by stereotyped UML associations. These associations can be bidirectional, have a defined semantics, and OCL can be used to define further specifications. Since the mapping rules are expressed as a MOF-compliant model, they can be accessed using a variety of approaches standardized by OMG (using API, XML, etc). Another important point about expressing transformation rules as models is that it will be possible to transform the transformation model itself! Thus, it will be possible to acquire new transformation rules by applying the general strategy of model transformations.

As we saw there are many different approaches to model transformations. Some

of them are relatively old and others appeared for just a few years ago. Sometimes different approaches can be combined by taking the best parts from each of them. In this paper will look at the MOF-based approaches. Model mapping languages for MOF-based transformations are an area for MDA technology adoptions. The current MOF Query/View/Transformation RFP requests technology submissions suited to the specification of the model mappings. In this paper we assume that there already exists a language that defines model mappings. The precise syntax of that imaginary language is not important because our main question is the possibility of combining PIMs and PSMs. It involves just identifying model elements that should be mapped and executing mapping rules given some formal notation for these rules.

Chapter 6

MOF-based PIM-to-PSM transformations

In the previous chapter we investigated the general concepts of model transformations. We saw that OMG proposed a new approach to transform models and to describe transformation rules. In this chapter we will look at the main kind of model transformations in MDA, namely PIM-to-PSM transformations. We will show the theory and supply concrete examples based on the Pet Shop sample application.

6.1 General concepts

Figure 6.1 on the next page illustrates the generic MDA pattern, by which a PIM can be transformed to a PSM. The transformations process can be seen as a function which receives four input parameters: the PIM which is the source model of the transformation, the platform model which describes concepts of the PSM, mappings that describe how to transform elements from the PIM into elements of the PSM, and some additional information which is necessary for transformation but which doesn't fit into other places. The return values of transformation function are: the PSM which is the target model of the transformation and the record of the transformation.

The PIM is prepared using one of many approaches defined by OMG. Among them are standard UML models, UML models extended with profiles, or models based on user defined metamodel. The most usual case is to define a profile that captures the main aspects of the system without going into any technical details. In chapter 3 on page 35 we saw a sample PIM model that describes the business functionality of the Pet Shop application. We defined a small profile that adds some semantics to the base UML meta-model (e.g. database independence). Regardless of what approach we choose to model a PIM, there must always be a formal metamodel that describes all concepts that can be found in the PIM. As we will see later, the metamodel of the modeling language may play a big role in the model transformation.

The platform model describes technical aspects specific to that platform. Often

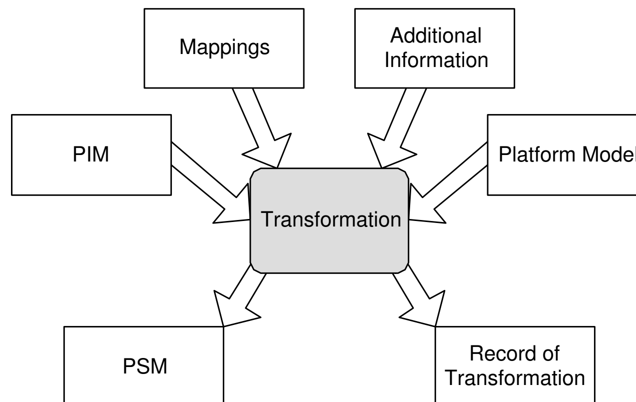


Figure 6.1: MDA transformation

this model is in the form of software and hardware manuals but in the case of MDA transformations the platform model must be formally described using one of the MOF-based notations. The platform model must describe as much as possible platform specific information to allow the transformation to use all strengths of that platform. In chapter 4 on page 49 we compared two platforms (J2EE and .NET) by looking at how they implement different concepts of a programming platform suited for building enterprise applications. We looked at runtime environments, typing systems, class libraries, component architectures, and other things. Ideally, all these aspects must be reflected in the platform model. If the platform model is incomplete then it will be no way to automate PIM-to-PSM transformations. A human architect will have to interfere into the transformation process and supply some information to resolve ambiguity problems, to define new mappings that use platform concepts missing in the platform model, and to omit some mappings from the transformation that otherwise are implied by the mapping rules.

A mapping is specified using some language to describe a transformation of one model to another. In general, a mapping tells us how element of certain type should be transformed into elements of another type. The description may be in a natural language, an algorithm in an action language, or in a model mapping language. A desirable quality of mapping language is portability. This enables use of mappings with different tools. Mapping languages may be based on XML Metadata Interface (XMI), Java Metadata Interface (JMI), and other standards. As discussed in the previous chapter, a mapping rule consists of left hand side (LHS) and right hand side (RHS). The LHS of a PIM-to-PSM mapping rule contains elements from the PIM while the RHS contains elements from the platform model. Both LHS and RHS may be composed of single model elements (e.g. types from class libraries, type attributes and methods, packages, relations between types, etc.), and/or patterns that represent specific design choices.

There are three kinds of mappings between elements from different models.

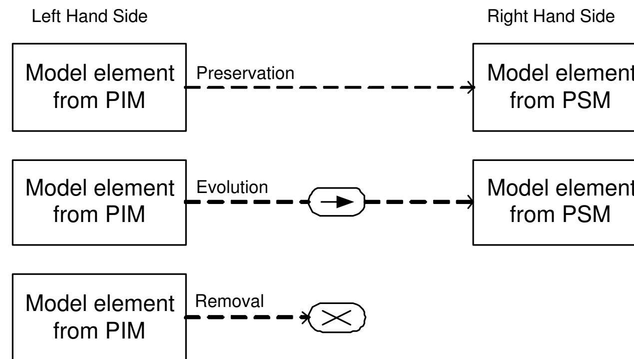


Figure 6.2: Three kinds of mappings

Figure 6.2 shows them: *preservation*, *evolution*, and *removal*. The preservation mapping copies elements from the LHS of the mapping rule to the target model without making any changes to these elements. This kind of mapping may be used when a PIM element that should be mapped to the PSM doesn't need to contain any platform specific information. In the evolution mapping the LHS of the mapping rule differs from the RHS. This is the most common kind of mapping. The third mapping kind is the removal mapping. It is used when the platform model doesn't have any concepts that correspond to the LHS of the mapping rule or when LHS represents a part of the pattern that maps to a smaller pattern in the target model.

The *Additional Information* box may contain many things that can guide a transformation. Often additional information will draw on the particular knowledge of the designer. This will be both knowledge of the application domain and the knowledge of the platform. Example of the additional information may be mapping specific information that it is difficult to depict in the transformation model (e.g. rule application scoping, rule application strategy, rule scheduling). Additional information may also contain application specific details that miss in the source PIM and cannot be derived by the mapping rules but that should be present in the target PSM.

The record of transformation includes a map from the element of the PIM to the corresponding elements of the PSM, and shows which parts of the mapping were used for each part of the transformation. This kind of information is useful for many purposes discussed in the previous chapter. An MDA modeling tool that keeps a record of transformation may keep a PIM and PSM in synchronization when changes are made to either.

6.2 Model type mappings

There are several different approaches to implement mappings between PIMs and PSMs. The first and the most simple approach is *model type mapping*. It specifies a mapping from any model built using types specified in the PIM lan-

guage to models expressed using types from a PSM language. A PIM is prepared using a platform independent modeling language. The architect chooses model elements of that language to build the PIM, according to the requirements of the application. These mappings may also specify mapping rules in terms of the instance values to be found in models expressed in the PIM language.

The PIM language that we used in defining PIM of the Pet Shop application is based upon the standard UML. There are only two kinds of types in this language: *classes* and *interfaces*. Both J2EE and .NET support these types. Thus, a class from our PIM should be mapped to a class in J2EE model and to a class in .NET model. The same applies to interface mappings. The names of classes and interfaces can be preserved under mappings. Unfortunately, the .NET language has three additional types: *delegates*, *structs*, and *enums* (see chapter 4 on page 49). This means that if we use model type mapping from our sample PIM language to .NET language then it will be impossible to use all .NET specific features.

This is the major limitation of model type mappings. When we compared J2EE and .NET platforms we pointed out that the .NET type system borrows many concepts from the Java type system and defines a large set of additional unique constructs. Thus, the .NET type system is semantically richer than the Java type system. The problem of PIM-to-.NET mappings stems from the fact that the standard UML language that we used for PIM modeling is much closer to the Java language than to the .NET CTS. As a consequence, many .NET specific concepts (e.g. extended sets of access and inheritance modifiers, *event* type member, etc) cannot be represented with the standard UML.

One of possible approaches to resolve this problem is to extend the PIM language to contain all features from the platform specific languages that can be a target of transformation. In the case of PIM-to-.NET model transformations the PIM language must contain concepts of delegates, structs, and enums in addition to common classes and interfaces. In PIM-to-J2EE mappings, enum can be mapped to a Java class that contains a set of individually defined constant values. Delegates can be mapped to appropriate collection of classes and interfaces which achieve similar to .NET delegates functionality (for example *observer* pattern). The situation with structs is more complex because the Java language doesn't have any analogue to .NET structs. The solution could be to map structs directly to Java classes.

This approach has several drawbacks. The PIM should be totally independent of the platform specific aspects. If we include delegates, structs, and enums in the PIM language, we must argue that these concepts are equally important for modeling object-oriented software just as classes and interfaces. We should not include these concepts in the PIM language if they can be represented with the existing concepts or if the PIM would not miss its expressiveness without these concepts. Another problem of this approach is that it can work only with our simple example where all target platforms (J2EE and .NET) are known in advance. But in a more general case we would not possess this knowledge and it will be impossible to consider all features of different platforms.

Type from PIM language	Mapping to type of	
	J2EE	**.NET**
boolean	boolean	System.Boolean (C# bool)
money	double	System.Decimal (C# decimal)
int	int	System.Int32 (C# int)
String	java.lang.String	System.String
DateTime	java.util.Date	System.DateTime
Address	Address	Address

Table 6.1: Mapping of basic data types from PIM to J2EE and .NET

However, the model type mapping approach works well with mappings of elements whose domain range is predefined and don't vary in the target platforms. Data types can be example of such elements. We can distinguish two kinds of data types: basic data types and user defined data types. Most of the programming languages have the same set of basic data types that includes doubles, integers, dates, strings, characters, etc. They can be directly embedded into programming languages or defined in the class libraries. The source of the user defined data types is the PIM model itself. These data types are application-specific. For example, in the PIM of Pet Shop sample application we defined *Address*, *CatalogItem*, *Order*, and others. Whenever names of these data types occur in definitions of types attributes and methods, they should be unchanged under PIM-to-PSM transformation.

Our PIM language contains a special *money* data type and we used values of this data type to store prices of catalog items in the PIM. All operations with money require extra-high precision and since different target platforms may have different computational capabilities, the mappings may also be different. For example, the .NET platform provides a *decimal* data type that defines 128-bit high-precision decimal numbers with 128 significant digits. In J2EE, the best match would be a *double* data type that is a 64-bit double-precision floating-point. Table 6.1 shows how some of the data types that we used in the PIM of the Pet Shop application can be transformed to the corresponding data types in J2EE PSM and .NET PSM.

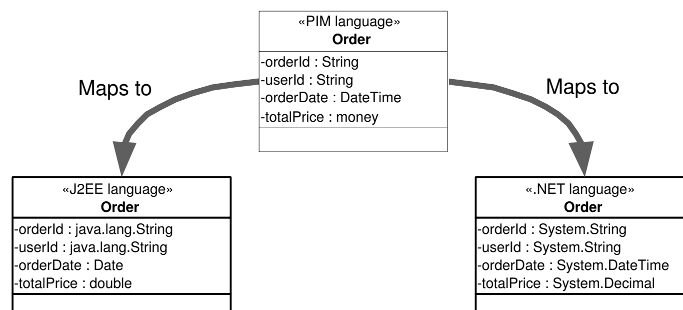


Figure 6.3: Class mapping

Figure 6.3 on the preceding page shows an example of mapping a model element from PIM to PSMs. This is a very simple mapping but it illustrates a model type mapping at work. Name of the class and its structure are unchanged under the transformation. Attributes also preserve their names and get new data types according to the table 6.1 on the facing page. Although class in this example doesn't have methods, method mapping would be quite similar to attribute mapping. The only difference is the description of parameter passing in the method definition. Our PIM language has three options for parameter passing: *in*, *out*, and *ref*. They are taken from the .NET CTS and PIM-to-.NET mappings would be straightforward. In J2EE model all parameter passing options would be mapped to *in*.

Mapping of collections deserves a special attention. Figure 6.4 shows a part of the Catalog module PIM where the *Page* class contains a collection of *CatalogItems*. The *Page* class has a method *getCatalogItems()* that returns the PIM data type *List*. When we map this method to J2EE and .NET platforms we must decide what kind of collection *List* represents. Collections may differ in many ways. The items that they include may be sorted or unsorted, unique or allow duplicates, they can be accessed directly or by keys. To indicate what particular kind of collection we want to get I extended the PIM language with three special tagged values for association relation: *isUnique*, *isOrdered*, and *isKeyed*. All these tagged values can be *true* or *false*. When we map the *List* to .NET and J2EE platforms we must look at the corresponding association relation and its tagged values.

Table 6.2 on the following page shows how different combinations of *isUnique*, *isOrdered*, and *isKeyed* decide which data type should be used at the models of target platforms. Data types that are written in italics represent interfaces meaning that several classes implementing this interface can be used. For some combinations of tagged values there are no data types with the exact meaning. In such cases we can map to a data type with the "closest" functionality. For example, J2EE doesn't provide data type for keyed collections with unique elements. We can use Map interface that supports keyed collections but allows duplicate elements. After this mapping the original PIM and the target J2EE PSM wouldn't be semantically equivalent, but the difference between them would be minimized. Alternatively, we can map to a user-defined Java class that implements Map interface and eliminates duplicate items.

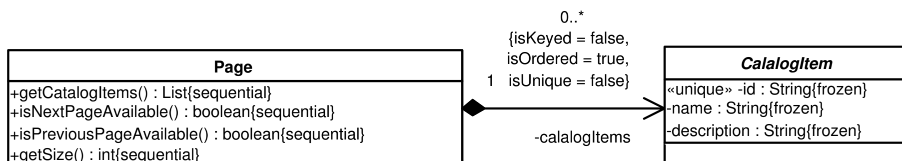


Figure 6.4: Mapping of collections

Association properties			**J2EE** (java.util)	**.NET** (System.Collections)
isUnique	**isOrdered**	**isKeyed**		
false	false	false	*Collection*	*ICollection*
false	false	true	*Map*	*IDictionary*
false	true	false	*List*	*IList*
false	true	true	*SortedMap*	SortedList
true	false	false	*Set*	none (default: *ICollection*)
true	false	true	none (default: *Map*)	none (default: *IDictionary*)
true	true	false	*SortedSet*	none (default: *IList*)
true	true	true	none (default: *SortedMap*)	none (default: SortedList)

Table 6.2: Truth table for PIM-to-J2EE and PIM-to-.NET Collections Mapping

Most of the data type entries in the table 6.2 are interfaces. In some cases it would be enough to know interface name, while in other cases we may be interested in a concrete class that implements that interface. Typically, there are many classes that implement a single interface. For example, in J2EE both *ArrayList* and *LinkedList* classes implement *List* interface. The language of our PIM is not sufficient to describe differences between these two classes. If we, however, want to map PIM List to one of these classes, then there are two opportunities. The simplest solutions is to define a default class and use it always. We may choose that whenever PIM-to-J2EE mapping produces Java List interface, we substitute it with ArrayList. Another solution is to determine whether the choice between ArrayList and LinkedList is based on platform-independent concept that would be meaningful to express in the PIM. If so, we can enhance PIM language to make it possible to express the distinction between these two collection types in PIMs in a platform-independent fashion.

The PIM-to-PSM mapping of collections is an example of model mapping that involves several model elements. We saw that in order to determine how the collection will be represented in the platform specific model we must consider not only the PIM type of the collection, but also tagged values of the corresponding association relations. In fact, most of the PIM-to-PSM mapping rules will consist of several model elements at both sides of the mapping relation. These kinds of rules map model elements according to patterns of type usage in the PIM. In the rest of this chapter we will look at many pattern mapping rules. When I show mapping rules I will follow some conventions that simplify understanding of the rules:

- Names of model elements in the PIM are surrounded with <brackets>.
- Names of model elements in the PSM are derived from names of model elements in the PIM. This allows identifying correspondences between particular model elements in the PIM and the PSM. For example, if there is a PIM model element with the name <Class> and a PSM model element with the name <Class>EJB then these model elements are related by the mapping rule.
- Two attributes with names <attr_1> and <attr_n> mean *all* class attributes. The same approach is used to express *all* functions.

- All attributes in the mapping rule have a generic datatype DT.
- All model elements that are not important part of the mapping rule are dropped from the mapping rule. For example, we use notation <op_1()> to express a function with arbitrary number of input parameters and arbitrary return value.

PIM-to-J2EE model type mappings

Figure 6.5 on page 103 shows the PIM-to-J2EE mapping rule for a PIM class with stereotype «Entity». The RHS of this mapping rule represents an entity bean which is a frequently used J2EE pattern known from the EJB component model. This pattern uses EJB specific stereotypes which are part of the J2EE platform language, thus reducing the size of the model. For example, we don't have to show the *javax.ejb.EntityBean* interface which the <Class>EJB class must implement because this information is implied by the «EJBImplementation» stereotype. The EJB model defines many mandatory methods (e.g. ejbLoad(), ejbStore(), ejbActivate(), ejbPassivate()) for different EJB classes and interfaces. We don't show these methods because they are also implied by EJB stereotypes.

Each PIM class with name <Class> and stereotype «Entity» should be mapped to a package with name <Class> and stereotype «EJBEntityBean» in the J2EE PSM. This package contains three elements — <Class>EJB class, <Class>LocalHome interface, and <Class>Local interface. Note that although there are both local and remote interfaces in the EJB model, our mapping rule suggests that only local interfaces are used in the target J2EE PSM. An attribute with stereotype «unique» in the PIM class <Class> is used as a primary key <key> in the entity bean <Class>EJB. This attribute is used in *create(in <key>)* and *findByPrimaryKey(in <key>)* methods in interface <Class>LocalHome. All attributes with stereotype «searchable» in the PIM class <Class> are used to generate finder methods in the J2EE interface <Class>LocalHome. For every attribute in the PIM class <Class> we generate *get* and *set* methods in the J2EE interface <Class>Local. An attribute with stereotype «unique» doesn't get a corresponding *set* method because primary keys should have read-only access.

Recall that there are two ways to persist entity beans: the bean-managed persistency (BMP) and the container-managed persistency (CMP). Our PIM «Entity» classes are intended to be independent of persistence mechanism. Thus, the mapping rule should generate entity beans with CMP. The <Class>EJB class is abstract because it doesn't specify how the data should be persisted. Of the same reason all operations that involve data access are also abstract. The EJB middleware would then generate a class that subclasses our <Class>EJB class and that implements a particular persistence mechanism. The tagged value *EJBPersistenceType* is used to specify BMP or CMP. Figure 6.6 on page 104 shows a complete example of mapping *Account* class from the PIM of the Customer module to the J2EE PSM of the same module.

Another nice feature of entity beans is that EJB containers provide many differ-

ent types of caching algorithms. Each of these algorithms has the same principle behind it: to reduce the frequency of ejbLoad() and ejbStore() methods, which are normally called on transactional boundaries. These caches can be set up using proprietary container tools or descriptors. No Java coding is required and that is why our PIM-to-J2EE mapping rule doesn't consider *canBeCached* tagged value of «Entity» PIM element.

The mapping rule that we just described can be used to map many classes with «Entity» stereotype from the PIM model of the Pet Shop application. But, unfortunately, there are some cases where this rule wouldn't work. Entity beans may have an arbitrary number of *finder* methods. However, at least one of them - ejbFindByPrimaryKey() - must always be defined. We saw that this method is generated from the «unique» attribute of the «Entity» class in the PIM model. In our PIM model of Pet Shop there are many «Entity» classes with missing «unique» attribute (Profile, Address, CreditCard, etc). How should we transform them?

An obvious solution is to define a new mapping rule that handles this special case. Figure 6.7 on page 105 shows how that rule looks like. It is very similar to that of figure 6.5 on page 103. The main difference is that there is no «unique» attribute in the «Entity» class of the PIM model. Thus, the <Class>EJB class of the J2EE model doesn't have a primary key field. The *create()* method of the J2EE interface <Class>LocalHome takes all attributes from the PIM class <Class> as input parameters. The *findByPrimaryKey()* method of the J2EE interface <Class>LocalHome takes one input parameter of type *Object* which plays a role of primary key. Figure 6.8 on page 106 illustrates this mapping rule.

We have now two mapping rules for mapping PIM «Entity» classes to J2EE PSM: in the figure 6.5 on page 103 (1) and in the figure 6.7 on page 105 (2). The second rule is a generalization of the first rule and it is possible to use rule (2) everywhere where rule (1) can be used. However, the transformation algorithm must always try to use the most specific rule first. In our case this would be rule (1). What particular rule is chosen by the transformation algorithm is the question of the rule application strategy. We can organize mapping rules hierarchically in a tree-like structure where the most general rule is the root of the tree. The transformation algorithm would then try to apply rules beginning with the leaf nodes of the tree and moving toward the root if not all mapping criteria are satisfied.

The PIM language of the Pet Shop defines another stereotype for classes — «Controller». This stereotype is applied to classes that represent some kind of action. In the EJB model, session beans serve exactly the same purpose. Thus, it is natural to map PIM classes with stereotype «Controller» to EJB session beans in the J2EE PSM. Figure 6.9 on page 107 shows an example of such mapping rule. This mapping rule is very similar to PIM-to-J2EE mapping rule of PIM «Entity» class because session beans and entity beans have much in common. Again we can see that the RHS of the mapping rule is a package with the same name as the original PIM «Controller» class. This package contains the implementation of the session bean <Class>EJB and two interfaces — local home interface and local interface.

There are two kinds of session beans in EJB model — stateful and stateless. Our PIM language doesn't provide any constructs that explicitly describe what kind of behavior (stateful or stateless) PIM «Controller» elements should support. We must make our decision based on some other available information. The first and the most simple approach is to use only stateless session beans in the target J2EE model. I used this approach in my PIM-to-J2EE mapping rules. Alternatively, we could examine PIM «Contoroller» class and try to guess the intended behavior. If, for example, PIM «Controller» class doesn't have any attributes or associations to other classes, then it has no possibility to store its state and the corresponding session bean should be stateless. If PIM «Controller» class has attributes or associations to other classes, then we can always map it to stateful session bean.

If a PIM «Controller» class has methods with stereotype «isTransactional», then the resulting session bean should support transactions. Actually, all methods of all enterprise beans must have transaction descriptors, but in the following discussion we will concentrate us only on session beans. Everything we discuss here equally applies to entity beans and message-driven beans as well. There are several ways how a session bean can support transactions. The key piece of information that decides what kind of transaction support a bean should use is *who* begins a transaction, *who* issues either commit or abort, and *when* each of these steps occur. Unfortunately, we can not deduce this information from our PIM model because our PIM language lacks this kind of expressiveness. This means that again we should define a default behavior that all session beans in target J2EE PSM will have.

There are three ways to demarcate transactions in session beans: *programmatically* (transaction logic must be programmed into application code), *declaratively* (the EJB container performs all transaction logic), and *client-initiated* (transactions are started and ended from the client code outside session bean). Our default choice is container-managed (declarative) transactions. We used a tagged value *EJBTransType* to describe the type of transaction support and assigned it *Container* value. For every method that participates in a transaction we must specify a *transaction attribute*. Transaction attribute tells the EJB container what role a method plays in the transaction. Its value can be *Required*, *RequiresNew*, *Supports*, *Mandatory*, *NotSupported*, or *Never*. All of these attributes describe different transactional behavior. We choose *RequiresNew* transaction attribute for every PIM method with «isTransactional» stereotype. This attribute means that a new transaction should be started every time a method is called. For every PIM method without «isTransactional» stereotype we choose *NotSupported* transaction attribute. Transaction attributes are stored in a special tagged value *EJBTransAttribute*. Since *NotSupported* transaction attribute is default, we don't show it in the target J2EE model.

PIM-to-.NET model type mappings

Mapping of the Pet Shop PIM to .NET PSM differs in many ways from the PIM-to-J2EE mapping. We saw that in PIM-to-J2EE mappings we took advantage of the EJB component model which provides session beans and entity beans.

In .NET platform, we can find concepts which resemble J2EE session beans, but there is no direct analogue to J2EE entity beans. Therefore, in PIM-to-.NET mappings we must apply different patterns that are specific to the .NET platform, yet resulting in a target Pet Shop PSM which has almost the same functionality as the J2EE PSM.

Figure 6.11 on page 109 shows a mapping rule that transforms a PIM pattern consisting of «Controller» class and one or more «Entity» classes which are instantiated by «Controller» class. We can see that PIM-to-.NET mapping rule produces packages in the target model just as the PIM-to-J2EE mapping rules. The major difference is that PIM-to-J2EE mapping rules always produced new packages with names derived from names of the original PIM elements. In PIM-to-.NET mapping rules the packages in the RHS have names that are independent of the names of the original PIM elements. This means that if a package already exists in the target .NET PSM (e.g. produced by some other mapping rule), then no new package should be created and the existing package should be used to place new elements.

The *BLL* package is used to store classes that are responsible for the business logic of the application. These classes can also contain methods that are used for interaction with clients of the application. The PIM «Controller» classes are placed in this package by PIM-to-.NET mapping rule without any structural changes. Only the «Controller» stereotype is removed from the class definition. The *Model* package is supposed to contain data types that are used both by clients of the application and the application itself. These data types are application specific. In our mapping rule we place classes derived from PIM «Entity» classes in *Model* package. Names of this classes get postfix *Info* because they don't implement any persistence logic. All other packages in the target .NET PSM contain classes that map data types from the *Model* package to external datasources.

For every PIM «Entity» class that is instantiated by PIM «Controller» class we should generate a class in the target .NET PSM that handles persistence logic. This class should contain basic database operations for insertion of new elements into database, deleting existing elements from the database, updating existing elements in the database, and finding existing elements in the database. Note that PIM «Entity» classes that are attributes of other PIM «Entity» classes and can not exist alone don't need to have own persistence mechanism (this is illustrated in figure 6.11 on page 109). Since a PIM «Entity» element can be database independent, there could be several corresponding data access classes in the target .NET PSM (one for each database).

We want to hide data access mechanism and, therefore, we create a separate interface which defines all data access methods and place it in the *IDAL* package. All data access classes should implement this interface. Such architectural decision allows referring to any of the data access classes using the same interface, thus hiding the persistence mechanism. This interface defines three standard database methods for insertion, updating, and deletion. In addition, there must be methods that select elements from the database. The *findByPrimaryKey(in <key>)* method selects a particular element based on the «unique» attribute of

the PIM «Entity» class. For every «searchable» attribute of the PIM «Entity» class we generate a corresponding finder method that returns a collection of elements.

For every database that is specified in the *dataSources* tagged value of the PIM «Entity» class we generate a package <DS>DAL in the target .NET PSM that stores data access classes for that PIM «Entity» class. Recall that ADO.NET has two ways to access database data: direct access (DataProvider object) and disconnected access (DataSet object). At this point we should decide what data access mechanism our data access classes should use. All necessary information can be found in the Pet Shop PIM. Our PIM language defines a *canBeCached* tagged value for «Entity» stereotype. If *canBeCached* tagged value is set to *yes* than we can use disconnected data access mechanism. The .NET infrastructure would then hold in-memory representation of «Entity»s database data. If *canBeCached* tagged value is set to *no* then we must use direct database access and check for new versions of «Entity»s data every time «Entity» object is requested. Tagged value *dataAccess* is used to denote data access mechanism in the target .NET PSM. Its value can be *DataProvider* or *DataSet*.

The last package in the .NET target model introduced by the PIM-to-.NET mapping rule in figure 6.11 on page 109 is called *DALFactory*. In this package we store factory classes that decide what datasource is used to map a particular «Entity» objects data. This decision can be made based on XML configuration files, Windows registry, or some other technique. The factory object instantiates a data access object and returns data access interface to the caller.

PIM «Controller» classes are not always mapped to .NET platform without any changes. If a PIM «Controller» class has methods that require support for transactions then it should be mapped as in the figure 6.13 on page 111. The original PIM «Controller» class is split into two classes: a class with stereotype «ServicedComponent» and an ordinary class. The first class uses functionality from the *System.EnterpriseServices* namespace and it contains only methods that require support for transactions. The second class contains all other methods. By splitting PIM «Controller» class into two we avoid the overhead of Enterprise Services. In «ServicedComponent» class we must specify the kind of transaction support. We use tagged value *TransactionOption* and assign it *RequiresNew* value.

Discussion of model type mapping rules

The above examples illustrated that we can use model type mappings to transform the PIM of the Pet Shop application to two different PSMs. Though neither of the resulting PSMs can be used to auto-generate the program code, they can serve as the basis for target implementations. Both J2EE PSM and .NET PSM contain many platform specific features that take advantage of the underlying technology. We may think that by using model type mappings we successfully achieved what we wanted — designed a common PIM which can be used to get arbitrary PSMs. The number of possible target PSMs is only limited by the number of platform models for which the mapping rules are defined.

If we take our PIM of the Pet Shop application, apply model type mapping rules for any platform on it, and look at the resulting PSM, then the chances that we like that model would not be big. What is the problem? The reason is that the fact that the PSM is workable doesn't ensure that the PSM is of the best possible quality. A good example can be our PIM-to-J2EE mapping of PIM «Entity» classes. We always map them to entity beans. There is nothing wrong with it and by using entity beans in J2EE PSM we achieve the desired functionality of mapping objects data to the database data. However, it is a known issue that the overuse of entity beans can have negative effects on the performance of the end system. It is a common practice to use entity beans only if it is absolutely necessary. If we can achieve entity beans functionality by using other technique, then we should do so. Many other examples of possible enhancements can be found both in .NET and J2EE PSMs of the Pet Shop application.

The main problem with model type mappings is that they are only capable of expressing transformations in terms of rules about things of one type in the PIM resulting in the generation of some thing(s) of some (one or more) type(s) in the PSM. However, without the ability for the architect add additional information to the transformation, the mappings will be deterministic, and will rely wholly on platform independent information to generate the PSM. We have already discussed the main limitations of model type mappings. Among them are:

- Insufficient level of abstraction of the PIM. Sometimes there is no distinction between very similar concepts in the PIM language.
- Not all distinctions between elements in a platform language can be expressed in a PIM because PIM should not contain any platform specific information.
- Sometimes there is not enough mapping rules to cover all concepts from the PIM language or the platform language.
- Sometimes we want to override predefined mapping rules based on our knowledge of the platform or the problem domain.

Because of these limitations, it is very unlikely that the PSM generated by model type mapping rules will be the best possible PSM for the given platform. Many platform specific design choices will never occur in the target PSM, that in some cases can damage the overall performance of the system. The worst thing that can happen is that PSM will not satisfy all business requirements of the application. If a modeler observes that the PSM can be improved, he can do manual changes to the PSM. This may be an acceptable approach, but not in the context of combining PIMs and PSMs. Our goal is to get rid of explicit PSMs and allow model modifications only on PIMs. PSMs must be as much as possible *read-only*. This is achievable only if PSMs can be computed by application of mapping rules on the original PIM. If the PSM is not satisfactory, then the changes must be done to PIM, mapping rules, or both.

We can conclude that the model type mappings work well if we are not after the best possible PSMs. This approach can be acceptable in some rare cases where we model small and not-critical applications. In all other cases we can't use

model type mappings, at least in their pure form. The problem can be solved if we find a way to interfere into the transformation process and substitute original mapping rules with new ones which are better suited for the particular PSM. Such approach must satisfy following criteria:

- Any changes that should guide the transformation must be done at the modeling phase and not during the transformation.
- Changes must be done to the PIM, but should not become a part of the PIM because they don't describe the system in a platform independent manner. Every PSM can need a separate set of changes.
- It should be possible to process these changes automatically in the same way as model type mapping rules.

A possible solution to guide a transformation is to define mapping rules on instances of particular model elements from PIM rather than on instances of model elements from PIMs meta-model. In such a way we can identify any element that needs a special treatment in the original PIM and define a special mapping rule that overrides an existing model type mapping rule for that element. This approach is called *model instance* mapping. We will look closely at model instance mappings in the next section. It is important to note that model instance mappings would rarely be used as the only rules for the transformation. Typically, model instance mappings would serve as a supplement to existing model type mappings.

6.3 Model instance mappings

The main idea of model instance mappings is to identify model elements in the PIM which should be transformed in a particular way, given the choice of a specific platform for the PSM. The main concept of model instance mapping is *mark*. A mark represents a concept in the PSM and is applied to an element of the PIM, to indicate how that element is to be transformed. Thus, marks are used together with associated model instance mapping rules. Usually, every PSM would use a distinct set of marks, but in some cases these sets may overlap for different PSMs. A mark that is used by several PSMs represents a concept which is common for several platforms, and it can be a good idea to include this concept in the PIM language.

Marks are usually used in two cases:

- to indicate non-functional and stylistic characteristics of the PSM, which can not be determined from information in the PIM (PIM is mainly concerned with functional characteristics of the system)
- to indicate functional characteristics of the PSM, which can not be determined from the PIM because of the insufficient level of abstraction of the PIM language

Although marks are applied to PIM elements, they are not part of the platform independent model. It should be easy to look at the PIM in its clear form

without any marks at all. It should also be possible to apply several sets of marks (one set for each PSM) on a PIM simultaneously. The marks can be thought of as being applied to a transparent layer placed over the PIM. The architect performs the following sequence of actions: he takes the platform independent model, decides for what particular platform he wants to get PSM, and marks the PIM with appropriate set of marks.

Marks are applied to certain types (or collections of types) in the PIM. Since all marks have specific meaning, they should be applied with care to make transformation make sense. Implicitly each type of model element in the PIM is only suitable for certain marks, which indicate what type of model element will be generated in the PSM. Transformations based on marking instances will either explicitly state which marks are suitable for which types in the PSM, or these type constraints will be implicitly understood by the user of the marks. For example, it makes no sense to mark association end in the PIM with an «Entity» mark. Improper marking of PIM elements may lead to an unexpected results after the transformation.

Marks are not part of the PIM language, but can be seen as an extension of the platform language for a particular PSM. Marks can come from different sources, which include:

- Types from a platform language, specified by classes, associations, or other model elements
- Roles from a platform language, for example, from patterns
- Stereotypes from UML profile for a platform language
- From any other sources, even not directly related to a platform language. Examples are:
  - Quality of service related aspects
  - Marks that indicate that model elements from PIM should not be mapped to PSM at all

In order for marks to be properly used, they may need to be structured, constrained or modeled. For example, a set of marks indicating mutually exclusive alternative mappings for a concept need to be grouped, so that an architect marking a PIM knows what the choices are, and that more than one of these marks cannot be applied to the same model element. Some marks may need parameters. It is a good practice to design sets of marks that do not depend on particular area of applications. Such sets of marks can be used with different mappings (different PIMs).

When an architect assigns a mark to a PIM element, that element should be transformed according to a model instance mapping rule associated with the mark and a model type mapping rule associated with that element. Sometimes model instance rules will completely override model type mapping rules, but more generally, these rules will be combined to achieve a proper transformation. For example, if a mark is applied to an attribute of a class then the class will be

transformed using model type mapping rule, except for the marked attribute, which will be transformed in a special way. The order in which different types of rules should be applied may be important in some cases.

An element of the PIM may be marked several times, with marks from different mappings. This indicates that the element plays a role in more than one mapping. When an element is marked in this way, it will be transformed according to each of the mappings. The result may be additional features of the resulting element(s) as well as the additional resulting elements in the PSM.

Using marks to define patterns

Previously in this chapter I said several times that mapping rules can be applied on patterns. I briefly defined a pattern as a collection of model elements which may occur on either part of the mapping rule. Now it is time to discuss patterns in more details because patterns are the most frequent sources for marks in the model type mappings.

Patterns are widely used in software development as well as in many other areas such as architecture, electrical engineering, or manufacturing. Each pattern describes a problem which occurs over and over again in a certain environment, and then describes the core of the solution to that problem. The main value of a pattern is that it can be freely used to solve similar kinds of problems without ever searching for the same solution twice. All patterns are rooted in practice. They are discovered by looking at what are the most common problems, what solutions to these problems exist, and which of these solutions give the best results. Of course, no solution is perfect and it can happen that none of the existing patterns can be applied for a particular problem without some modifications. However, identifying the pattern that is closest to a specific problem can be a good start in solving that problem.

There exist hundreds of patterns for solving software related problems. Most of patterns have well-known names which can be used to refer to a specific pattern unambiguously. We can also define our own names for patterns that we invent ourselves. In model type mapping the most natural way to associate a model element with a pattern is to use stereotypes or tagged values. The main benefit of embedding pattern information into a PIM is that we can significantly reduce the size of the PIM. Instead of showing a whole pattern (which may consist of a large number of elements) in a PIM, we can show just a single model element which is stereotyped with the patterns name. This element will be mapped to a full scale pattern in the PSM.

Unfortunately, it is not always possible to decide what patterns should be used at the PIM modeling phase. In this case we can use pattern names to mark a PIM just before we map it to the PSM. Among the reasons for marking PIM elements with pattern names are:

Insufficient level of abstraction of the PIM language Sometimes the PIM language is not powerful enough to distinguish between very similar problems which can be solved with different patterns. Thus, the only way to

apply a right mapping rule is to guide the mapping manually by assigning appropriate marks to the PIM elements.

Incomplete definition of the mapping rules Each pattern is relatively independent, but patterns are not isolated from each other. Often one pattern leads to another pattern or one occurs only if another is around. The definition of mapping rules may fail to identify such dependences.

Now we will look at an example of marking the PIM of the Pet Shop application. We will examine how we can improve the J2EE PSM of the Catalog module if we use patterns.

In web site user views, catalog data are displayed in much the same way as they are stored — as tables. In addition, catalog data is read-only in the Pet Shop web site. In situations when relational data are being accessed and used in tabular form, and especially when access is read-only, it's preferable to access the data relationally, using *Data Access Object* pattern (DAO) to encapsulate the data access mechanism. In such situations, using entity beans incurs a performance penalty while providing little or no additional value. Because we don't want to represent catalog entries as entity beans, their persistence must be managed in code. The DAO pattern uses JDBC to read data directly from a database and it hides the differences in SQL implementations between vendors. The DAO pattern also makes it easy to change how the catalog access its entries.

Figure 6.15 on page 112 illustrates how a PIM element marked with *J2EE:DAO* mark should be mapped to the J2EE PSM. The marks name consists of two parts. The first part (J2EE) indicates the platforms name and the second part of the marks name (DAO) indicates the patterns name. It is necessary to include the platforms name in the marks name because by following this naming convention we can create distinct sets of marks for different platforms. If there are marks with identical names for different platforms then it would be impossible to put marks for several platforms in the same PIM because it would be no way to find out to which mapping each mark belongs. For example, if we define a mark with name *DAO* for both J2EE and .NET and apply this mark to a PIM element, then we would be forced to apply DAO pattern in mappings for both platforms, though the original intention may be to use DAO pattern only in J2EE mapping.

The mark J2EE:DAO can be applied only to a PIM «Entity» element. Any other place (e.g. PIM «Controller» element) this mark will make no sense because DAO pattern can only be used for accessing persistent data. In our case only the PIM «Entity» element has such semantics. Figure 6.15 on page 112 shows that the mapping rule requires that the marked PIM «Entity» element must be instantiated by some other element (it doesn't matter what particular kind element).

The marked PIM <ClassE> element is mapped without any structural changed to a *Model* package in the J2EE PSM. The PIM <ClassC> is mapped to J2EE PSM according to the model type mapping rules specified for that PIM element.

If none of the existing mapping rules match this element, then it is just copied to the J2EE PSM without any changes. All model elements that constitute the DAO pattern are gathered in the package named DAO in the J2EE PSM. These elements are: <ClassC>DAOFactory class, <ClassC>DAO interface, and one DS<ClassC>DAO class for each datasource DS specified in the PIM «Entity» elements *DataSource* tagged value. The <ClassC>DAO interface duplicates all methods from the PIM <ClassC> element and makes the DAO "pluggable". This interface is created by <ClassC>DAOFactory class which has a single method *getDAO()*. <ClassC>DAO interface is related with <ClassC> class by a composite aggregation. Different sources of PIM «Entity» elements data are accommodated by creating different DS<ClassC>DAO classes that implement <ClassC>DAO interface.

Figure 6.16 on page 113 shows how the mapping rule that we just described can be applied to the PIM of the Catalog module. This figure should be intuitive. The only interesting point that may need some explanation is the mapping of the PIM CatalogManager element. This element has a «Controller» stereotype and it should be mapped to a stateless session bean (according to the model type mapping rule defined previously in this chapter).

Other reasons to mark a PIM

Sometimes an architect may want to make some manual changes to a PIM just before the mapping to a particular platform takes place. These changes should not become a permanent part of a PIM because they are not independent of the choice of a platform. It may sound strange that in certain circumstances there is a need to make modifications to the PIM itself because the PIM must (by definition) describe the problem in a platform independent fashion. However, there are some cases when the problem that the PIM describes looks differently from viewpoints of different platforms. There are several explanations to this fact. First of all, the poor level of abstraction of the PIM language may prevent to describe the system precisely. We have already discussed this problem and seen some examples.

The second reason is that only the CIM (Computational Independent Model) describes the problem in a completely platform independent way. All other kinds of PIMs rely more or less on platform specific aspects which may not always be compatible with the target platforms of the PIM-to-PSM mappings. For example, if we model a PIM with class diagrams (use aspects of object orientation), we can have problems with mapping this PIM to a platform where the concept of classes is missing (e.g. Fortran, COBOL).

The next reason is that it is often a very challenging task to model a PIM that describes all possible relevant aspects of a system. Moreover, in many cases we don't need a PIM with this level of details because this PIM would become very complicated and difficult to understand and maintain. In some situations it can also be difficult to decide what aspects of the system are relevant and should be represented in a PIM. We may think that certain properties of a system don't have any impact on the implementation and then we decide not to include these properties in a PIM. Later it can turn out that this assumption holds for several

platforms, but not for all.

Marks can be used to cope with the discussed problem. We can see at a PIM-to-PSM transformation as a sequence of two transformations: PIM-to-PSM_1 and PSM_1-to-PSM_2 where PSM_1 is a modified PIM and PSM_2 is a target PSM. The first transformation will wholly rely on model instance mappings with marks and the second transformation will use both model instance mappings with marks and model type mappings.

My next example of using marks with model instance mappings is again based on the Catalog module PIM, but at this time we are interested in enchancing the .NET PSM. Figure 3.5 on page 45 shows the PIM of the Catalog module. The main purpose of the Catalog module is to provide a read-only access to list of categories, products, and items. It should be possible for the clients of the Catalog module to view catalog data page by page. To achieve this functionality I defined a class *Page* which can return catalog data in portions of a fixed size. This class a is necessary part of our PIM if we want to allow different clients to access the product catalog in a similar way (this is one of the requirements to the Catalog module). Suppose now that for some reason we decide that the .NET version of the Catalog module should be accessable only by web clients. I will show that this little piece of information which was not available when we modeled the Catalog module PIM can infuence PIM-to-.NET mapping.

Web clients in .NET are implemented with ASP.NET. ASP.NET provides a hierarchy of *WebControl* classes which can render themselves either as a single HTML element or as a combination of HTML elements based on a control state. Many of these controls provide such useful features as state manegement, data validation, and data binding. While data can be bound to any control, several controls simplify the common case of data presentation, including *DataGrid*, *Repeater*, and *DataList* controls. These controls implicitely provide such custom paging capabilities as page count or page navigation. It is possible to bind many different types of data sources, including simple collection classes and data readers connected to a database. We can see now that we don't need a special *Page* class in the Catalog module PIM because ASP.NET provides all the paging functionality that we need.

When we map the Catalog module PIM to .NET PSM we must begin with some modifications of the PIM which will remove the *Page* class from the PIM. I defined a special mark *.NET:Remove* that can be applied to any model element in a PIM. This marks says that the marked element must be removed from the model, but it says nothing about how the removal operation must be performed. Thus, we must define model instance mapping rules for the mark. If we don't define mapping rules for a partiuclar mark then the application of this mark wouldn't have any effect on the transformation. There can be several different instanse type mapping rules for the *.NET:Remove* mark because in most cases when we remove an element from the model, we must make adjustments to all model elements related to the element being removed. Since there can be different kinds of relations between model elements, the adjustments can also be different.

Figure 6.17 on page 114 shows one of possible instanse type mapping rules which can be associated with the *.NET:Remove* mark. The LHS of the mapping rule represents a pattern of three elements. In this mapping rule the marked element (<ClassB>) is related by dependency relationship to one element (<ClassA>) and by association relationship to another element (<ClassC>). When we remove the <ClassB> element from the model we must remove both existing relationships and establish a new dependency relationship between <ClassA> and <ClassC>. This is done by taking the original dependency relationship and making its end pointing to <ClassB> to point to <ClassC>. All other properties of the original dependency relationship (e.g. name, stereotype, tagged values) should remain unchanged.

We may also need to make some adjustments to the elements <ClassA> and <ClassC>. If the PIMs <ClassA> has functions which signatures contain the <ClassB> element (e.g. types of input parameters or type of return values) then the function signatures must be changed. All occurances of the <ClassB> element in the function definitions must be substituted with new elements according to the following rules. If the *End2* end of the original association relationship has multiplicity 1 then all occurances of <ClassB> become <ClassC>. Otherwise, we must use the truth table 6.2 on page 88 to determine the new type. All properties of the PIMs <ClassC> element which signatures contain the <ClassB> element must be removed from the <ClassC>. Figure 6.18 on page 114 shows an example of PIM-to-.NET mapping of the Catalog module PIM marked with *.NET:Remove* mark.

Marks can be used not only to remove elements from the PIM but also to add new elements. Take a look at the PIM of the Customer module (figure 3.4 on page 43) and the corresponding PIM-to-.NET mapping (figure 6.12 on page 110) based on the model type mapping rule. We can see that the .NET PSM of the Customer module provides no way to update customer information. The clients of the .NET Customer module can ask the *SignOnManager* object for the customer information encapsulated in the *AccountInfo* object. All customer information is read-only for the clients because *AccountInfo* class has no functionality to persist its state. Note that this problem is specific for the PIM-to-.NET mapping. In the PIM-to-J2EE mapping the customer information is represented by an entity bean which can update its state. It means that there is no problem with the PIM itself, there is a problem with model type mapping rules. This example shows that by using model type mapping rules we can loose some information from the PIM.

To deal with this problem we can use marks. The main question that arises is how can we enchance our .NET PSM of the Customer module (what new elements can we add to the PSM and where should we add them)? If we look closely at the .NET PSM of the Customer module we shall see that all the functionaly to persist customer information already lies in the PSM. Interface *IAccount* which we use to create new customers and find existing customers has a function *update(in param: AccountInfo)*. This function is essentially what we need. Unfortunatelly, it is never used in our .NET PSM. Classes that implement *IAccount* interface are used by the *SignOnManager* class which in turn is accessible by clients. Thus, all we need to do is to create a new function *updateAc-*

count(in account: AccountInfo) in *SignOnManager* class. Clients would then be able to update customer information by calling this function. Figures 6.19 on page 115 and 6.20 on page 115 show both the model instance mapping rule definition that can create a new function and an application of this rule.

Discussion of model instance mapping rules

The above examples showed that the model instance mapping rules can be used to guide PIM-to-PSM transformations in a very flexible way. Whenever model type mapping rules fail to produce a good PSM, an architect can define some marks which identify how a particular PIM element or a collection of PIM elements should be transformed. These marks in combination with instance type mapping rules can achieve virtually any effect. They can add new elements to the PIM, remove existing elements from the PIM, override existing model type mapping rules, and create new mapping rules.

Marks can be of two types: platform-specific and problem-specific. The distinction between these two types of marks is not always clear and sometimes a mark can be both platform-specific and problem-specific. Both types of marks are related to a particular platform but they serve different roles under the transformation. Platform-specific marks help to identify elements from the platform language (e.g. types or platform-specific patterns) which should be used to map PIM elements. Platform-specific marks should be carefully defined and categorized because they can be reused in many PIM-to-PSM mappings for a particular platform. The *J2EE:DAO* mark is a typical example of a platform-specific mark. It identifies a well-known pattern which can be used not only in the mapping of the Pet Shop PIM but also in other mappings.

Problem-specific marks are related to a specific PIM. They are used to make structural modifications to the PIM and sometimes they are designed for use in a single PIM-to-PSM mapping. Examples of problem-specific marks can be *.NET:Remove* which removes PIM elements or *.NET:addFunction()* which adds new functions to PIM classes. Although these two marks can be used in multiple PIM-to-PSM mappings, we can also think of other problem-specific marks which can be applied to only one PIM-to-PSM mapping.

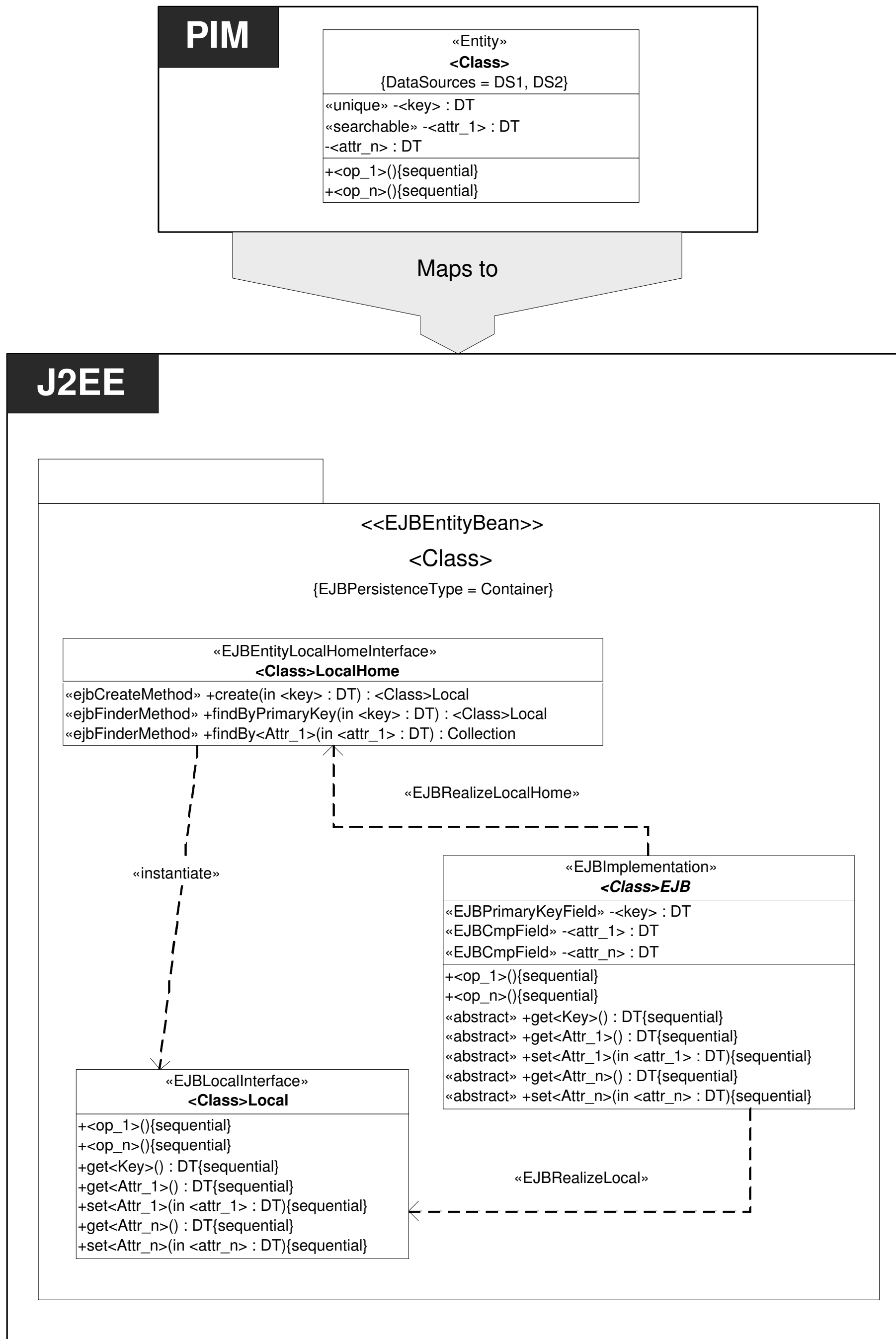


Figure 6.5: PIM-to-J2EE model type mapping of «Entity» class

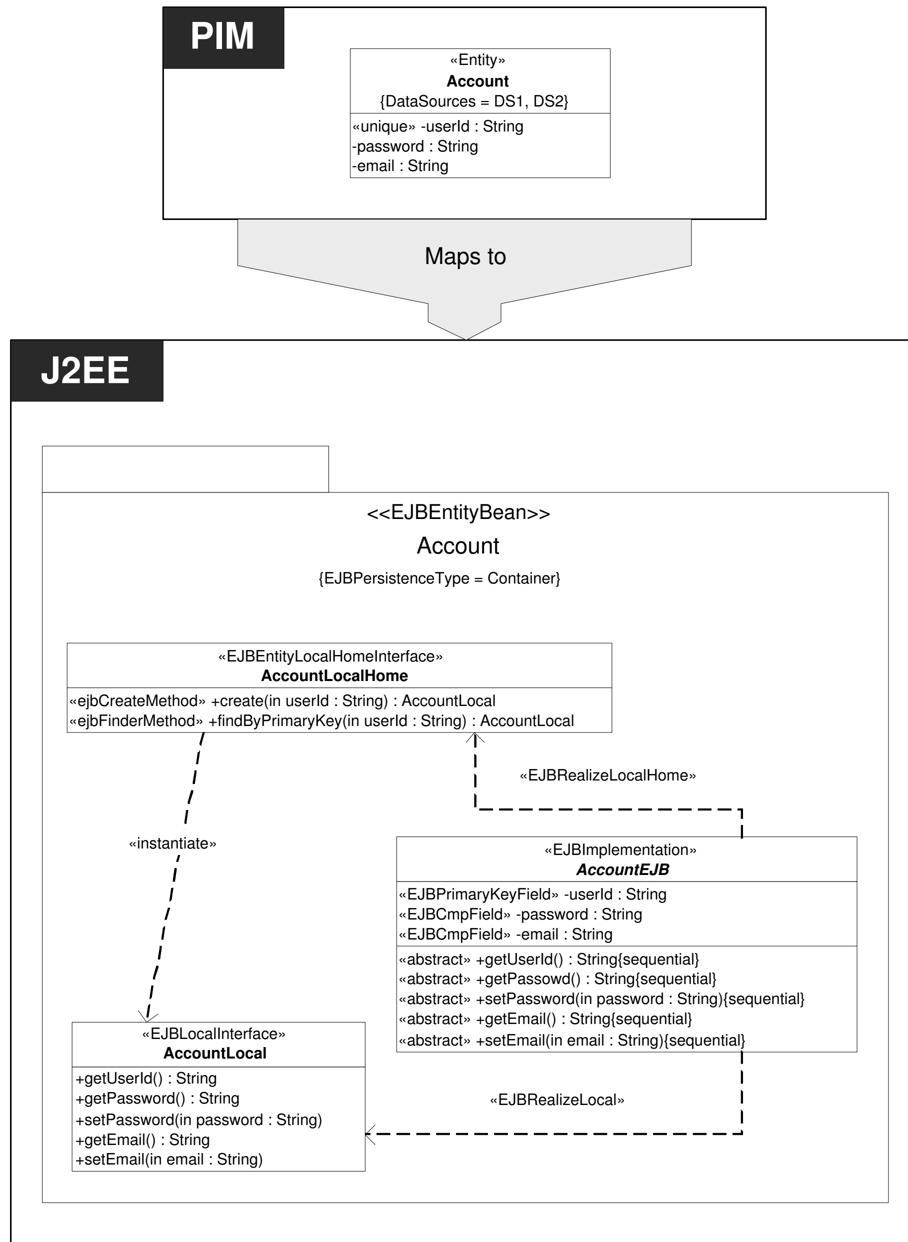


Figure 6.6: Example of PIM-to-J2EE model type mapping of PIM «Entity» class

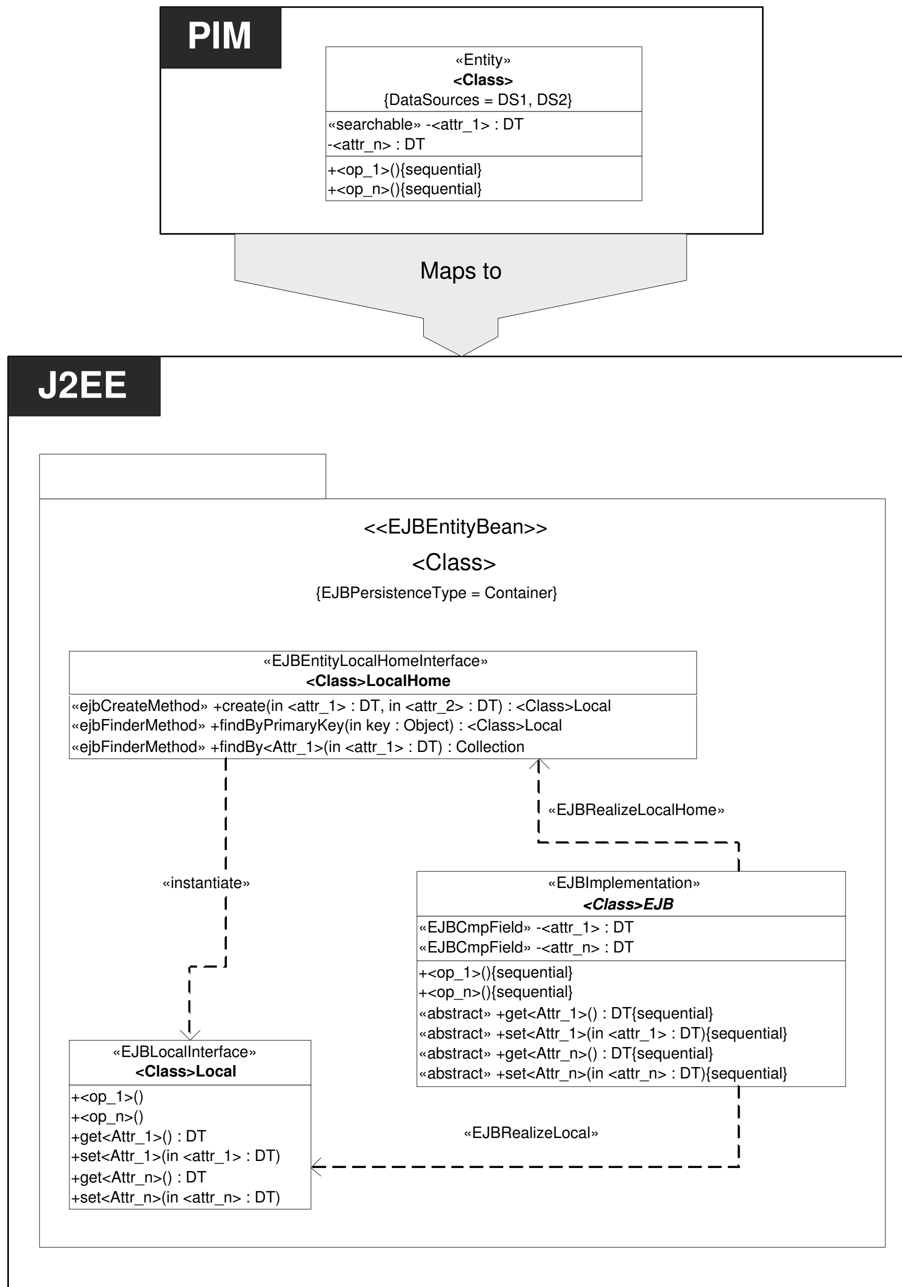


Figure 6.7: PIM-to-J2EE model type mapping of PIM «Entity» class with missing «unique» attribute

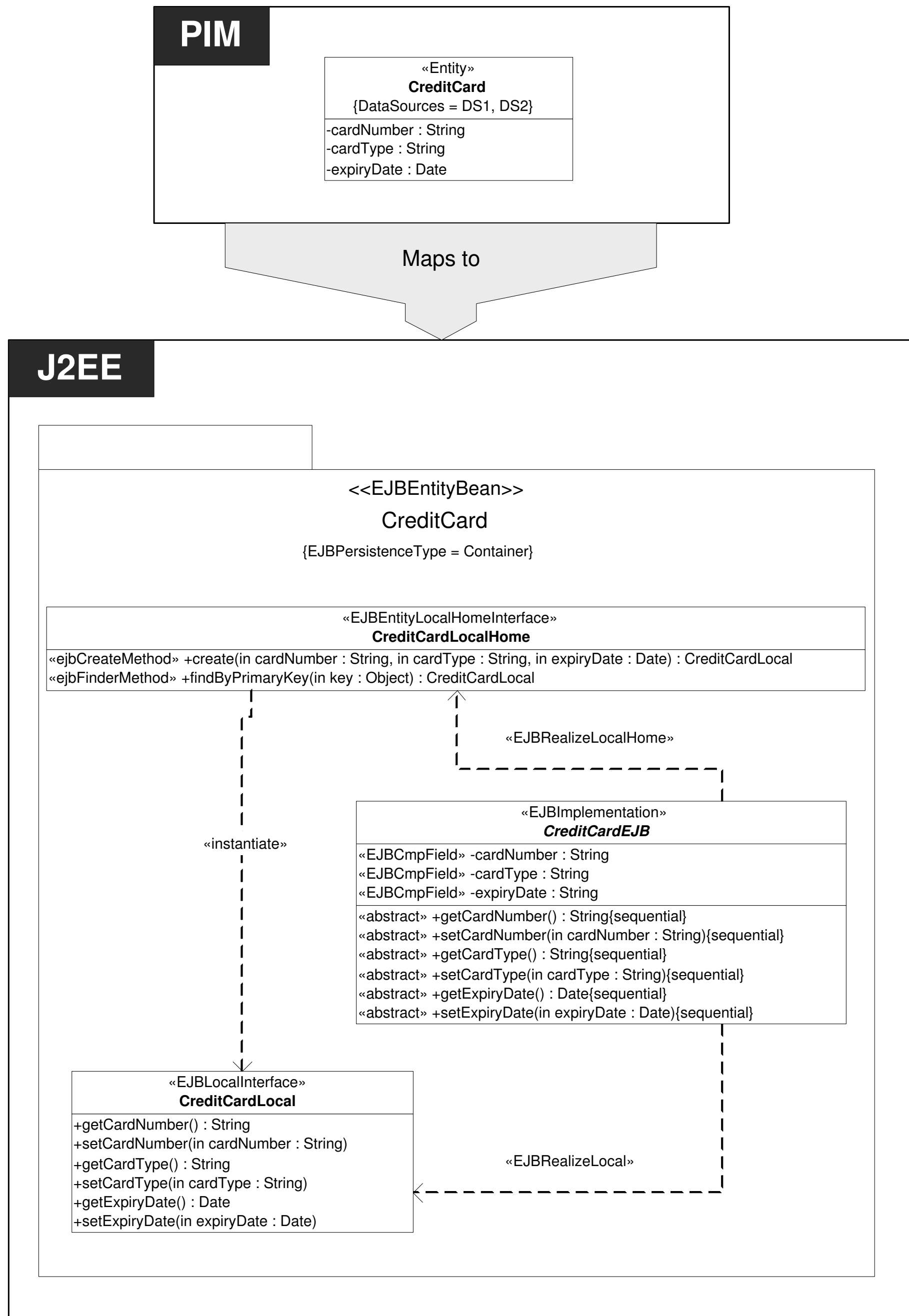


Figure 6.8: Example of PIM-to-J2EE model type mapping of PIM «Entity» class with missing «unique» attribute

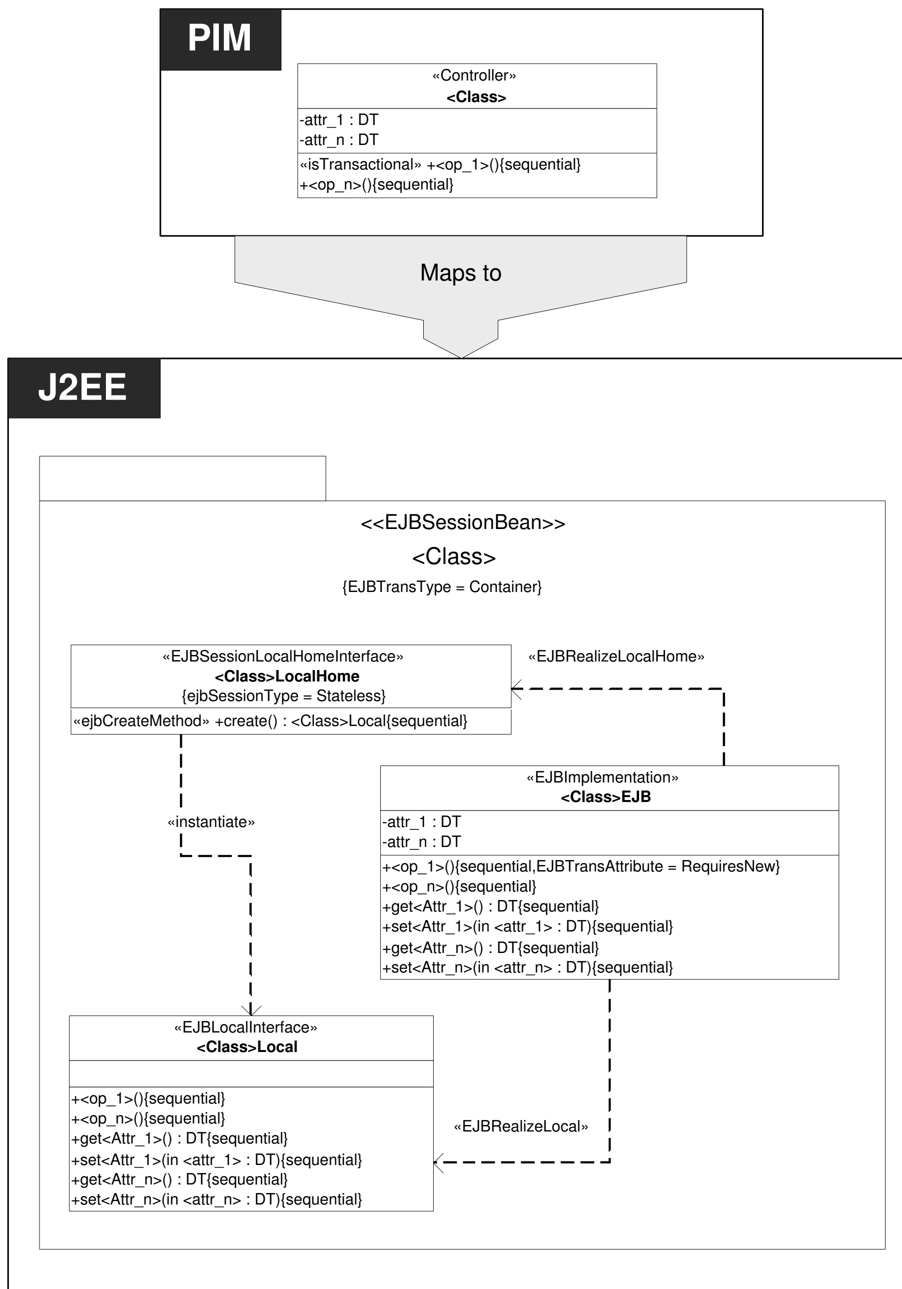


Figure 6.9: PIM-to-J2EE model type mapping of PIM «Controller» class

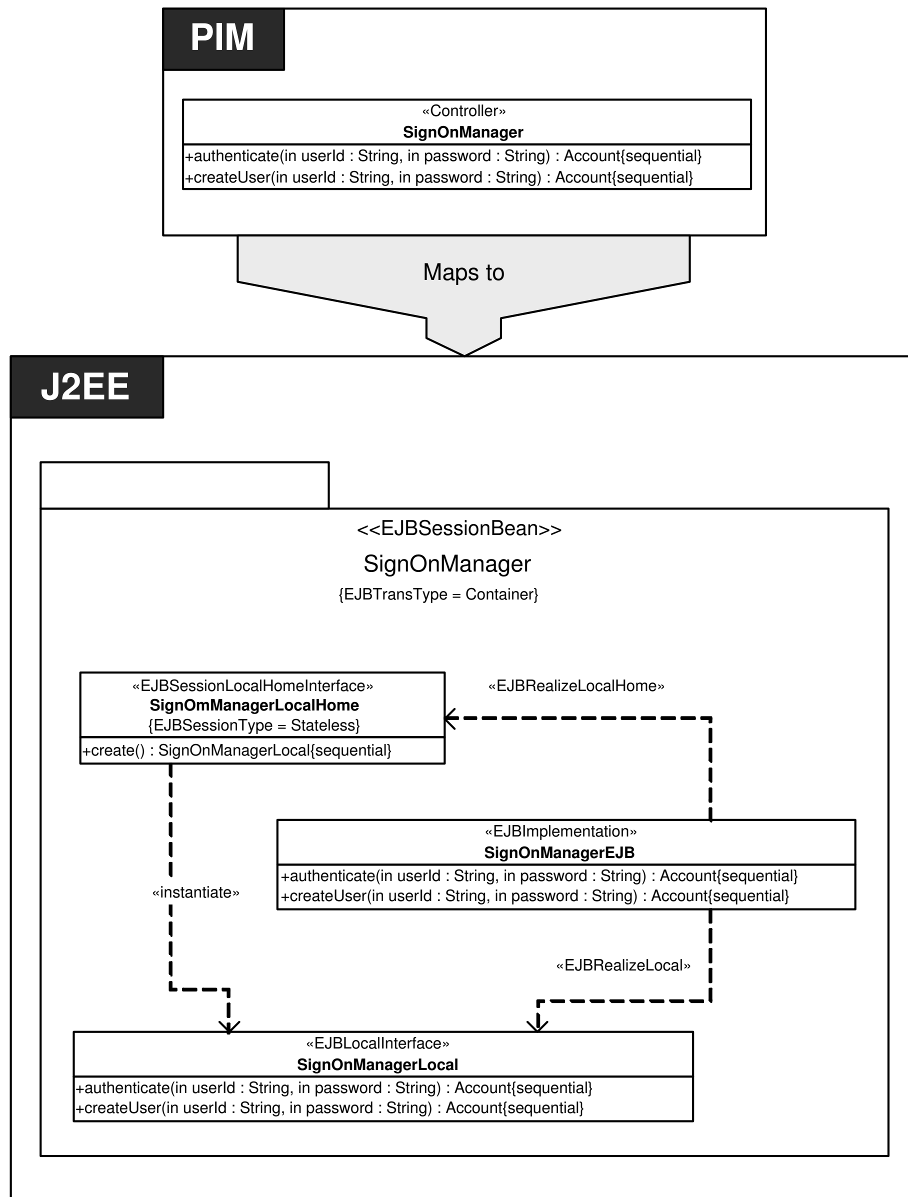


Figure 6.10: Example of PIM-to-J2EE model type mapping of PIM «Controller» class

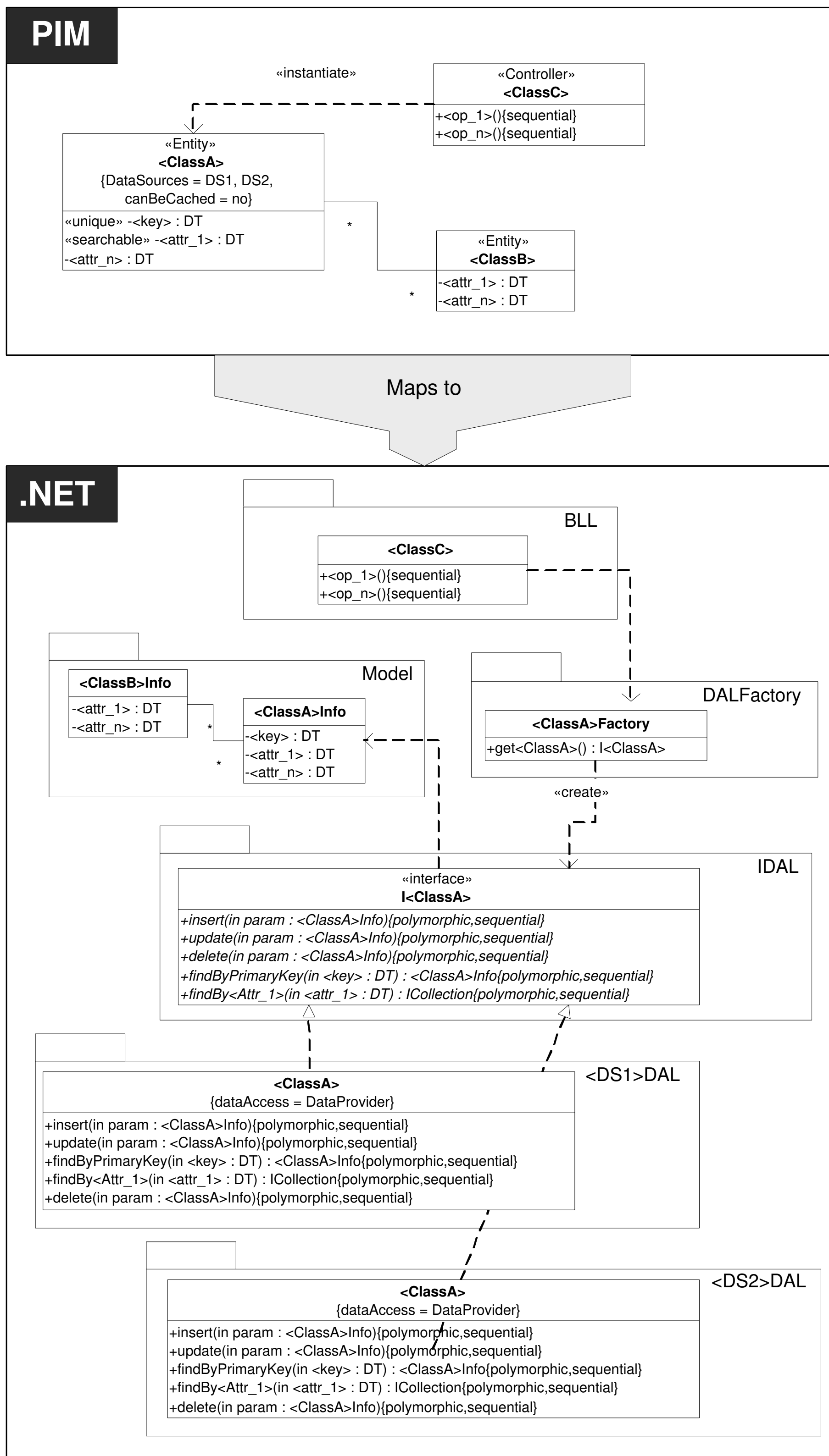


Figure 6.11: PIM-to-.NET model type mapping of PIM «Controller» and «Entity» classes

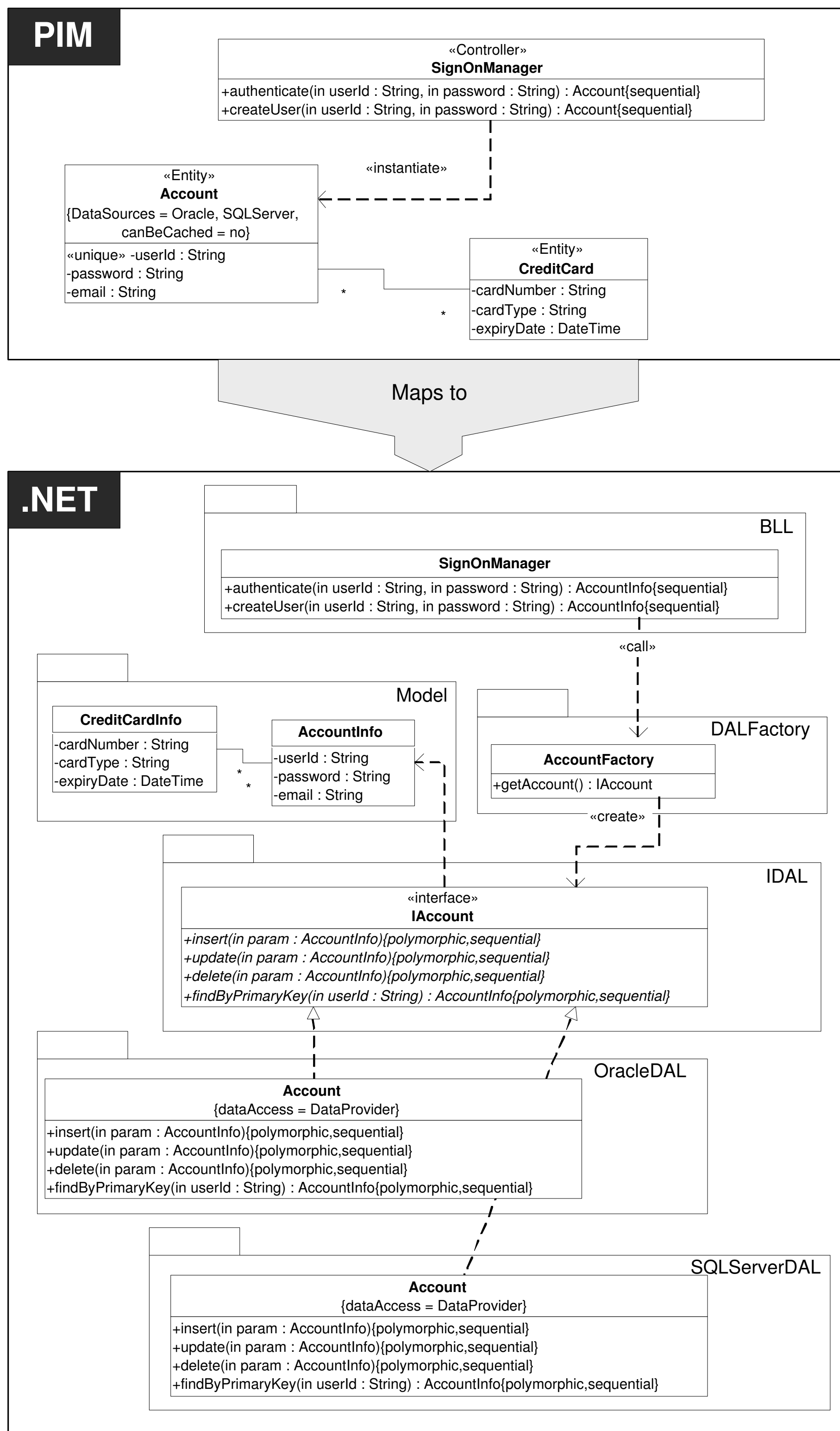


Figure 6.12: Example of PIM-to-.NET model type mapping of PIM «Controller» and «Entity» classes

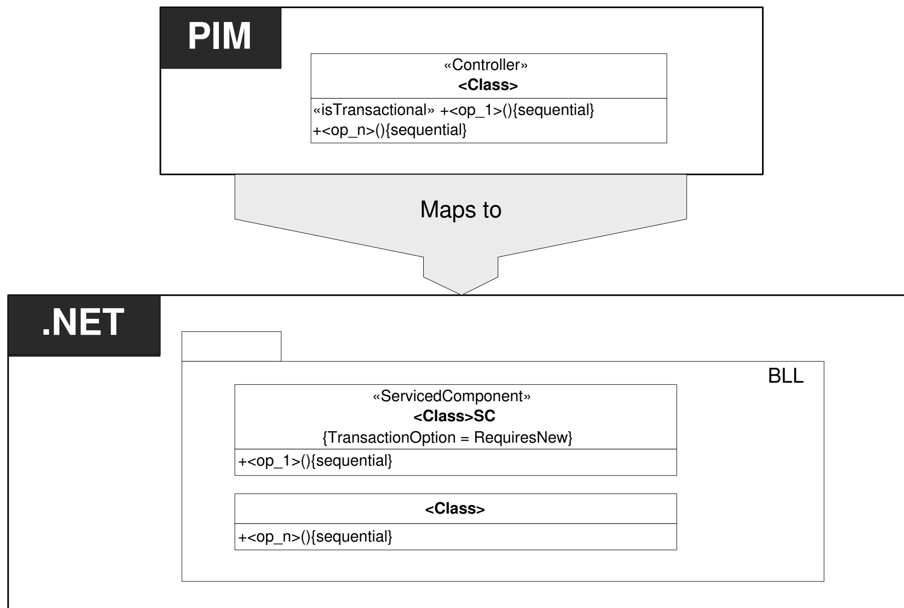


Figure 6.13: PIM-to-.NET model type mapping of PIM «Controller» class that supports transactions

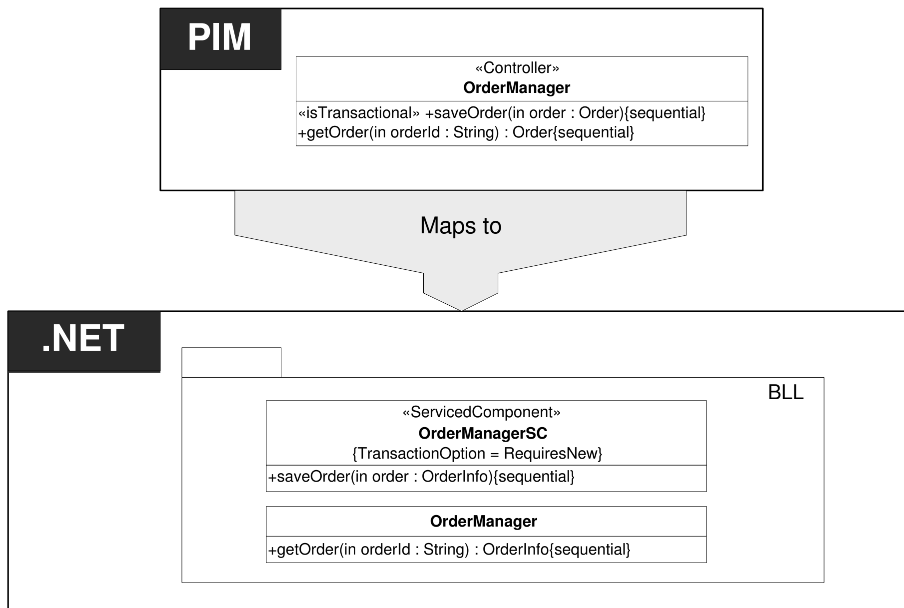


Figure 6.14: Example of PIM-to-.NET model type mapping of PIM «Controller» class that supports transactions

Figure 6.15: PIM-to-J2EE model instance mapping of a PIM element marked with *J2EE:DAO* mark

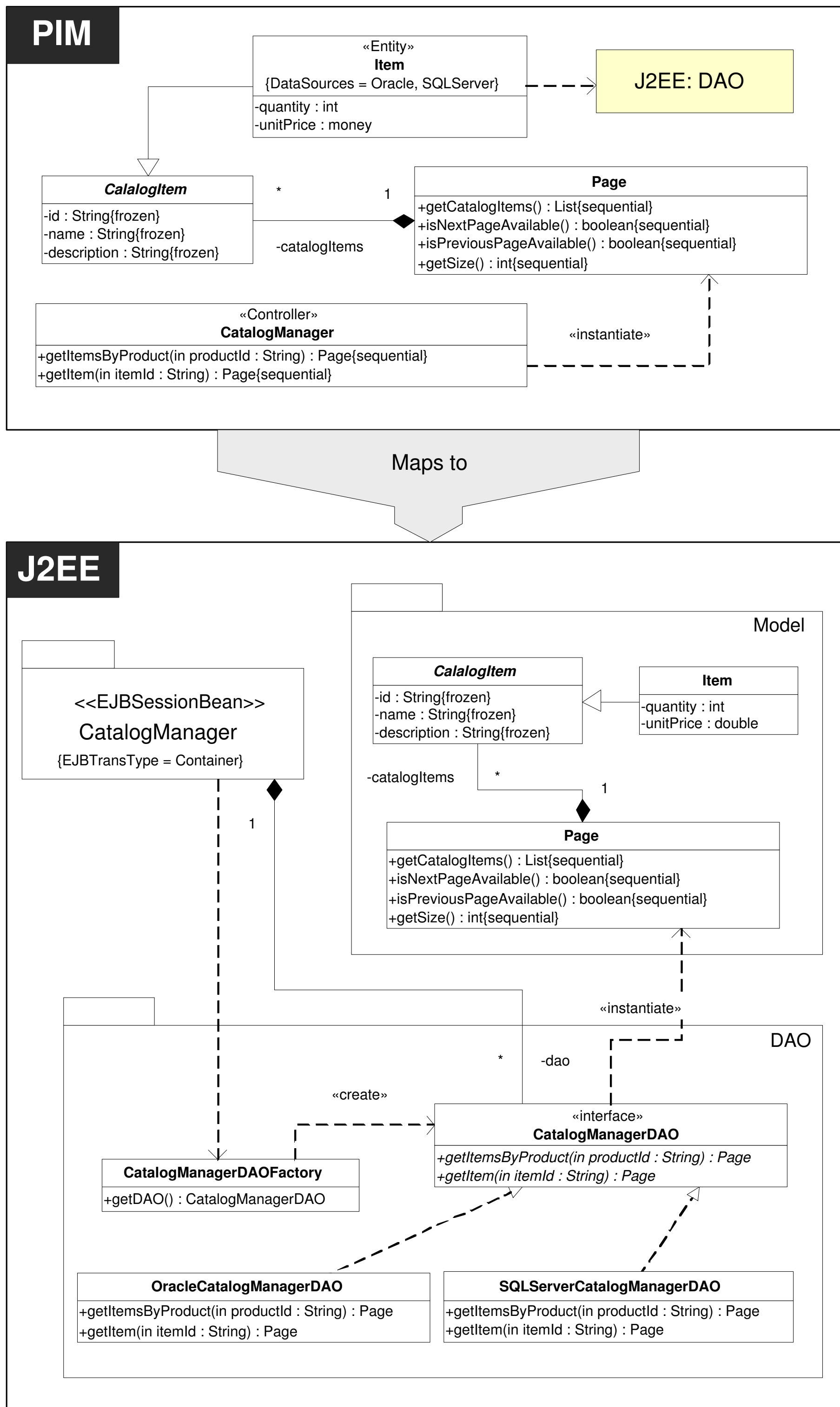


Figure 6.16: Example of PIM-to-J2EE model instance mapping of a PIM element marked with *J2EE:DAO* mark

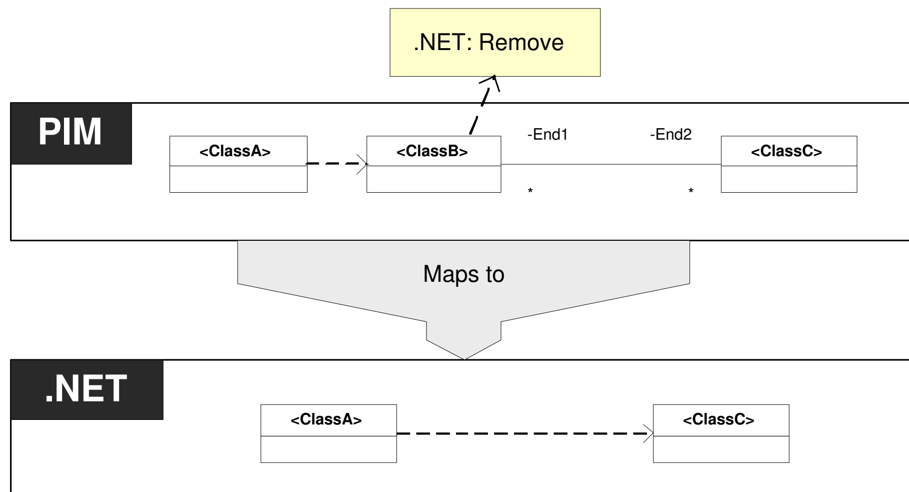


Figure 6.17: PIM-to-.NET instance type mapping of a PIM element marked with *.NET:Remove* mark

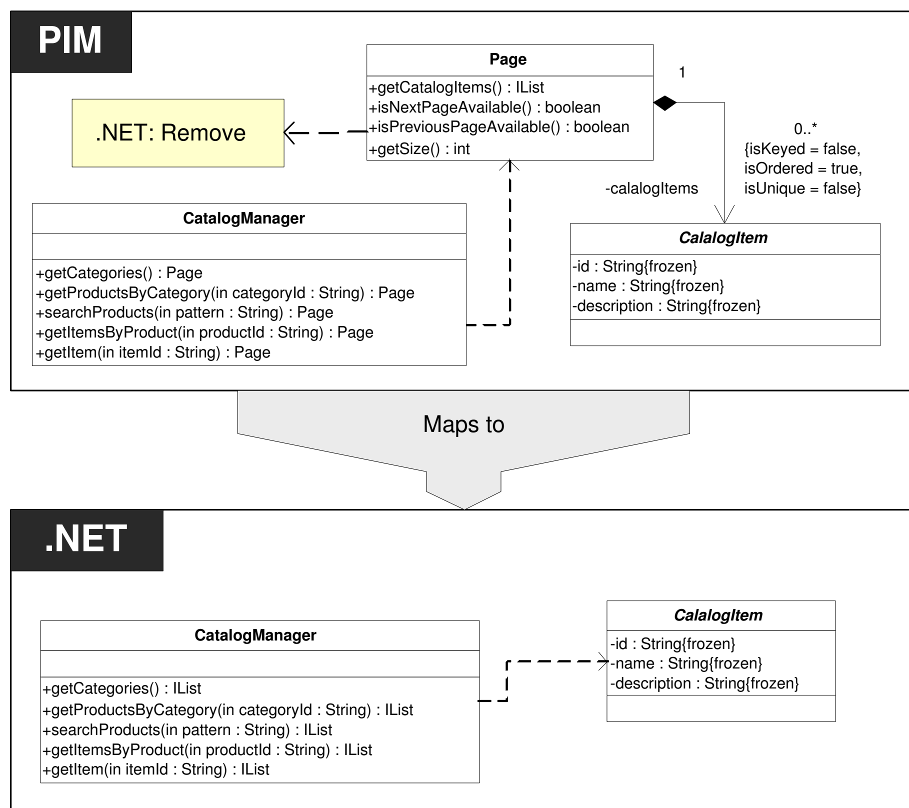


Figure 6.18: Example of PIM-to-.NET model instance mapping of a PIM element marked with *.NET:Remove* mark

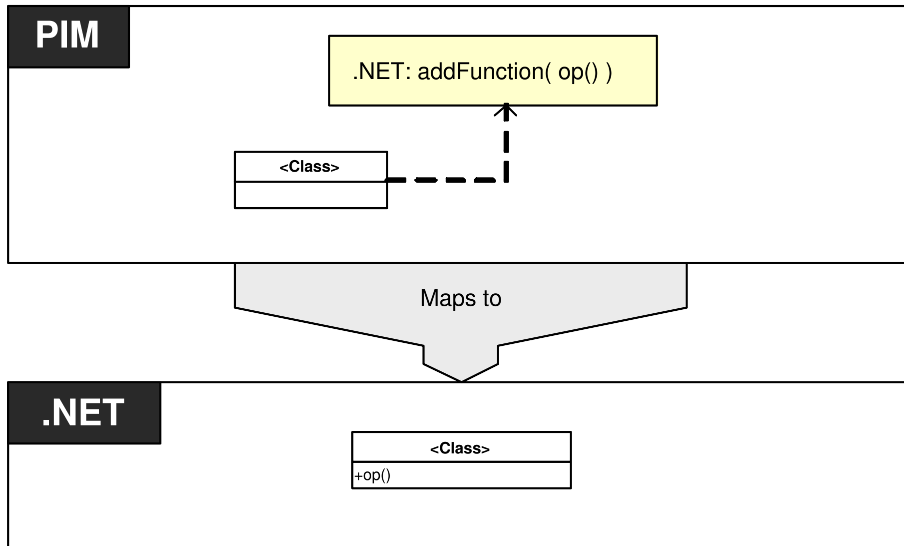


Figure 6.19: PIM-to-.NET instance type mapping of a PIM element marked with *.NET:addFunction()* mark

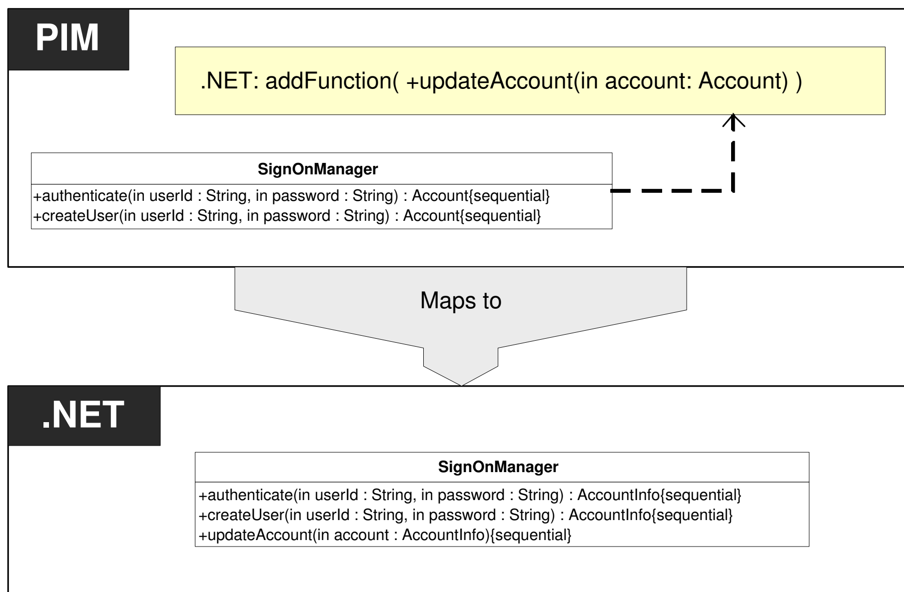


Figure 6.20: Example of PIM-to-.NET model instance mapping of a PIM element marked with *.NET:addFunction()* mark

Chapter 7

Combining PIM and PSM

In the previous chapters we discussed many important features of the MDA such as MOF, PIMs, PSMs, and mapping rules. To illustrate the theory I contructed PIM of the Pet Shop application and showed how it can be mapped to the two different PSMs: one for the J2EE platform and one for the .NET platform. Now it is time to look the main question of this master thesis — combining PIMs and PSMs.

7.1 What is a *Combined PIM-PSM*?

The MDA defines a framework for system architecture development methodologies. The main characteristic of all MDA-compliant methodologies is that every step of the system development process results in producing a model. Different system architecture development methodologies can use different kinds of models depending on what aspects of the system development they should cover. For example, there exist methodologies that consider business modeling as a necessary starting point of every system development project and methodologies that don't demand business modeling. However, if a system architecture development methodology is to be called MDA-compliant, it must include at least two kinds of models: Platform Independent Models and Platform Specific Models.

A PIM expresses only business functionality and behavior which are independent of the choice of technology. A PSM contains all the information from the PIM in addition to the platform related issues. A platform may be many things. The Pet Shop PIM that I defined in chapter 3 on page 35 is independent of the *platform for development of enterprise applications*, which is a set of technologies that abstract differences in hardware, OS, DBMS, and middleware from the application developer. Examples of platforms that I used in the Pet Shop PSMs were .NET and J2EE.

The PIM modeling phase precedes the PSM modeling phase. When an architect defines a PIM for some system, he must choose at least one target platform for the systems implementation and model the systems PSM for that platform. It follows that there exists a relationship between a PIM and any PSM that is

based on that PIM. Given a pair of a PIM and one if its PSMs, it must be possible to identify correspondences between different parts of these models. Relationships between a PIM and its PSMs can be either formally defined and documented or they can be totally informal. In the latter case, an architect can explain relationships but it is hard to prove their correctness. The only way to define relationships formally is to express them in the form of mapping rules.

The PSM modeling is usually the last phase of an MDA-compliant system architecture development methodology. PSMs can be further transformed to program code which is also a model but code generation is typically not a part of the MDA process because of two reasons. First of all, PSMs are not necessarily on a higher level of abstaction than the the code. The second reason is that we don't know how to define rules for mapping PSMs to program code. Usually, the code generation is done manually in an ad hoc manner and many elements of the program code can't be directly related to the PSM.

A major benefit of formal mapping rules is that they can allow to automate the PSM generation. We can construct a model transformer that takes a PIM and a set of mapping rules for a particular platform as an input and produces a corresponding PSM as an output. If the set of formal mapping rules is missing or incomplete then no such transformer would be possible and all PSM modeling must be performed manually by the architect. The ability to perform PIM-to-PSM transformations automatically is extremely valuable because it makes the presence of explicit PSMs unnecessary. When we have multiple explicit PSMs we come into problem of model synchronizations. Every time we make a change to a PIM we must immediately update all corresponding PSMs to reflect this change. This may be a difficult and time-consuming task if we have many PSMs for a single PIM. On the other hand, if we can generate PSMs automatically then the PIM-PSM synchronization will also be automatic.

The automatic PIM-PSM synchronization allows to make changes to both models at only one place as they were the same model. In other words, it allows to *combine* a PIM and the corresponding PSMs. The possibility of combining a PIM and the corresponding PSMs is the main question of this master thesis. In order to answer it we must decide whether the automatic PIM-PSM synchronizations is possible or not. The latter question can in turn be answered if we decide whether it is possible to define formal mapping rules between an arbitrary pair of a PIM and a corresponding PSM or not. If we can't define formal mapping rules that allow automatic synchronization between models then there is not way to combine a PIMs with its PSMs.

7.2 Is it possible to combine PIMs and PSMs?

Now I will discuss the most crucial aspects of PIM-to-PSM mapping rules based on the experience I gained in the previous chapters. I will define the main requirements for the existence of combined PIM-PSM and look how these requirements are satisfied by the MDA.

7.2.1 Requirement 1: Existence of standardized and predefined languages for PIMs and PSMs

Mapping rules define how elements in one model are represented in another model after completed mapping. A mapping rule consists of two parts: left hand side (LHS) and right hand side (RHS). The LHS represents an element or a collection of elements from the original model and the RHS represents an element or a collection of elements from the target model. Both RHS and LHS of a formal mapping rule must be expressed using formal grammars which can be different. The grammars used to express LHS and RHS of a mapping rule must be the same as the grammars that are used to express model elements in the original and the target model. In the case of PIM-to-PSM mapping rules it means that both PIM and PSM must be defined using formal grammars. Grammars that are used in modeling are called meta-models. So, the first requirement for the existence of formal PIM-to-PSM mapping rules is the ability to define meta-models that describe languages of respectively PIM and PSM.

The Meta Object Facility (MOF) is an OMG standard defining a common, abstract language for the specification of meta-models. MOF is distinctly object-oriented in nature. It defines the essential elements, syntax, and structure of meta-models that are used to construct object-oriented models of various systems. MOF serves as a common model for many meta-models where the most important are meta-models of UML and CWM. UML contains many useful constructs for expressing different software-related issues. UML can be extended with stereotypes and tagged values to meet the special needs of any particular application. Because of its formal definition UML can be the language of choice for modeling PIMs and PSMs. UML is especially good for modeling object-oriented systems. For modeling other types of systems it is always possible to define a new modeling language based on MOF meta-model. Thus, the requirement for formal languages that should be used for modeling PIMs and PSMs is easily satisfied.

When we want to model a PIM we can either try to find an existing modeling language or define a new modeling language if none of the existing languages are good enough. By a PIM modeling language I mean the base UML extended with a profile which covers all important aspects from that PIM. Obviously, using an existing modeling language spares much time and increases the value of PIM modeling. Thus, the effectiveness of PIM-to-PSM mapping rules depends on the variety of the predefined and standardized PIM languages and their correctness and completeness. Recall that there can be two types of PIMs: Computational Independent Models (CIMs) and PIMs that include some aspects of technology even though platform-specific details are absent. We need modeling languages for both types of PIMs. Moreover, sometimes we may need to combine several PIM languages to model a single PIM.

Figure 7.1 on the facing page shows the MDA logo that we already saw in the chapter about MDA. This logo describes the MDA infrastructure. The outermost and the largest ring, dominating the diagram with its compass points, depicts the various vertical markets or domains whose facilities will make the bulk of the MDA. Languages describing different aspects from these markets

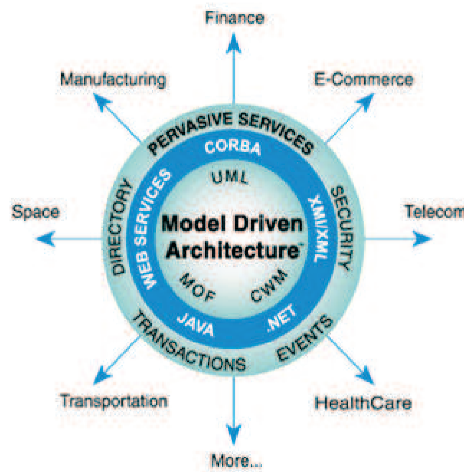


Figure 7.1: Model Driven Architecture

are the languages that must be used in modeling CIMs. For example, we may think of a CIM language for manufacturing with concepts of Product Data Management and CAD/CAM interoperability, a CIM language for healthcare with concepts for patient identification and medical records access, or a CIM language for B2B and B2C e-Commerce describing financial functions. Each of these languages must be a specialized UML profile which is capable of identifying as much as possible different aspects from the problem domain. Not all of the markets depicted in the OMGs logo have good UML profiles. Besides, there is a large amount of important markets which are not in the OMGs logo and which don't have own UML profiles at all. We still have a lot of work to do if we want to have a predefined modeling language for any CIM.

PIMs that include some aspects of technology (we can call them the second-level PIMs where CIMs are the first-level PIMs) need own modeling languages. Languages of the second-level PIMs may include such concepts as persistence, transactionality, security level, or even configuration information. OMG has already defined a number of standardized UML profiles for specific technological environments. Examples are the UML profile for Enterprise Distributed Object Computing (EDOC) which can be used at modeling collaborations of all types and the UML profile for Enterprise Application Integration (EAI), specialized for modeling applications based on asynchronous communication. The main problem with the existing profiles (e.g. UML profile for EDOC) is that they are very big and poorly structured. Another problem is that for many important technological environments there are still no standardized profiles.

We can conclude that we have a relatively bad support for modeling PIMs. However, the situation may become better in several years if the OMG continues to work in direction of defining standardized UML profiles for different markets and technological environments. Now we will look at the PSM modeling languages. Many of the problems related to PIM languages hold for PSM languages. But

there are also some problems that are specific to PSM languages.

A language used in modeling a PSM is a platform model. Since a platform can be many things, a platform model may describe many different aspects of a platform. For example, a DBMS platform may specify the data model (e.g. network, relational, object-oriented), the data languages (data definition, data manipulation, data query), transaction management (e.g. flat, chained, nested), the client-server architecture (file server, page server, object server) and many more. A model of a platform for development of enterprise applications must at least describe the programming language (e.g. type system, type libraries, APIs), the middleware services (e.g. security, persistence, transactionality), the layering of applications (e.g. what layers are supported and how they are supported), and the best practices of using that platform (e.g. patterns). The platform model defines an UML profile which must be used in modeling PSMs for that platform. An example of a platform model is the UML profile for CORBA standardized by OMG in 2000. This profile can be used in modeling PSMs for CORBA.

The main problem with platform models is that very few of them are defined and standardized. In my examples I used J2EE and .NET platforms. The .NET platform doesn't have any standardized UML profile at all. In the case of the J2EE platform we can use the standardized UML profile for EJB, but EJB is just a small part of the J2EE specification. The problem of missing PSM languages may resemble the problem of missing PIM languages. However, there is an important difference. We can expect OMG to standardize many of the PIM languages in the coming years because OMG is interested in a soon adoption of the MDA. On the other side, OMG has little to do with standardizing of the PSM languages, especially for proprietary platforms.

Another difference between PIM languages and PSM languages is that PSM languages change more frequently than PIM languages. For example, if we define an UML profile for healthcare, then we can use that profile for a long time because the healthcare market is relatively stable in its requirements. The situation with technology platforms is different. They develop very quickly and new features appear almost all the time. This means that UML profiles for technology platforms must be updated much more frequently than UML profiles for PIMs. Theoretically, every release of each platforms new version must contain an updated UML profile for that platform.

We can see that we lack standardized modeling languages for both PIMs and PSMs. This significantly lessens the current value of MDA because spending much time on defining own modeling languages increases the total time of development. As a consequence, the cost of production increases too. When I defined the Pet Shop PIM, I had to invent my own little UML profile because none of the existing UML profiles was good enough for my purposes. It was not a very difficult task to identify the most important platform-independent aspects that were relevant for the Pet Shop application. However, it took some time to structure these aspects, associate them with new stereotypes, give meaningful names to the stereotypes, and decide what tagged values can be used with this stereotypes.

When we think about combined PIM-PSMs then the absence of standardized languages for PIMs and PSMs gets an additional negative consequence. PIM-to-PSM mapping rules depend on the PIM language and the PSM language because mapping rules use concepts from both these languages. If we had standardized languages for modeling PIMs and PSMs then we could also standardize mapping rules. These rules could then be reused in different PIM-to-PSM mappings and the time of development of a combined PIM-PSM would be reduced. We can conclude that although absence of the standardized languages for PIMs and PSMs is not a critical problem, it makes the process of developing combined PIM-PSMs a lot harder.

7.2.2 Requirement 2: Existence of expressive and unambiguous mapping rules

PIM-to-PSM mapping rules define how an element or a collection of elements from a PIM should be mapped to a PSM. It is possible to make a combined PIM-PSM for an arbitrary PIM only if for each model element in the PIM language there exists a mapping rule with this model element in its LHS. If we want to get a PIM-to-PSM mapping of the best possible quality, then for each model element from the PSM language there must exist at least one mapping rule with this element in its RHS. The last requirement ensures that the PSM can use all properties of the underlying platform. If the PIM language or the PSM language supports patterns (collections of model elements which act as single units with certain responsibilities) then there must be a mapping rule for each such pattern. If we don't have a standardized set of mapping rules and define our own mapping rules for a particular PIM-to-PSM mapping then we must define at least one mapping rule for each element (and pattern) from the PIM and for those elements (and patterns) from the PSM language that we want to use in the PSM.

We should avoid making ambiguous mapping rules where the LHS of one mapping rule is a part of the LHS of another mapping rule. Consider the following situation: we have a mapping rule with the LHS consisting of a single model element *E*, a mapping rule with the LHS consisting of a pattern which contains model element *E*, and a PIM which matches both mapping rules. What mapping rule should we use? There are at least two approaches. The simplest one can be called *Same Always*. We can identify all mapping rules that may conflict with each other and choose one of them to be primary. Whenever a conflicting situation occurs we should always use the primary rule. The disadvantage of this approach is that the many of the mapping rules may never be used and the PSM will not be able to take the maximum advantage of the platform. The second approach is to define a rule application strategy where the choice of the mapping rule depends on some specified factors.

So far we have discussed mapping rules that fall under the category of *model type mapping rules*. These rules uniquely identify how each concept from the PIM language is mapped to the concept(s) from the PSM language. We can make a combined PIM-PSM using just model type mapping rules only under the following condition: the PIM language must be at least at the same level

of abstraction as the PSM language. Otherwise, mapping rules may become ambiguous. Suppose that the PIM language contains a single concept C and the PSM language contains two distinct concepts C_1 and C_2 which are variations of C. We will have two model type mapping rules: $C \rightarrow C_1$ and $C \rightarrow C_2$. It will be impossible for the transformer to determine which rule should be used to map C. We saw this situation several times in the Pet Shop PIM-to-J2EE and PIM-to-.NET mappings. The general solution to this problem is to decrease the level of abstraction of the PIM language.

It is unrealistic to expect PIM languages to be at the same level of abstraction as PSM languages because PIMs should be free of all platform specific details. However, many of the concepts from the PSM languages can be included in the PIM languages. Usually, we want to be able to transform a particular PIM to several different PSMs. These PSMs are based on platforms that are rarely totally independent of each other and that share some common information. The amount of common information across different platforms depends on how close these platforms stay to each other. In chapter 4 on page 49 I compared the .NET platform with the J2EE platform. I showed that these platforms are very similar. They share a lot of common concepts such as almost identical type systems, type libraries, and enterprise services. We can consider these concepts to be platform independent in respect to J2EE and .NET and include them in the PIM language.

The platform-specific concepts can also be expressed in a platform-independent manner but this is a little trickier task. We can try to find such platform-specific concepts from different platforms that although being not directly comparable still have the same role in certain circumstances. In many cases the functionality of a platform-specific concept from one platform can be achieved by using a pattern (a combination of concepts) from another platform. A good example of such platform-specific concept is the *entity bean* in the J2EE platform. It has the *entity* role (a unit capable to map itself to the database data). Although there are no entity beans in the .NET platform, we can easily find a pattern of .NET concepts with almost the same role. Therefore we can include the concept *entity* in the PIM language. It will have one-to-one correspondences with the J2EE-specific concept *entity bean* and some .NET-specific pattern.

The point is that by carefully examining related platforms we can find out that they are actually not so different from each other. All of them are designed to solve the same kinds of problems and the only difference between related platforms is how they solve these problems. The same problem can be easily solved in one platform by using simple platform-specific concepts, while in other platforms we may need to use complicated patterns. This observation means that it is theoretically possible to make a PIM language with the same level of abstraction as the PSM language and to define unique model type mapping rules. It can be done in the following way:

- All the problems that the platform is intended to solve must be identified and included in the PIM language.
- For each problem we make a model type mapping rule with the LHS consisting of a problem description expressed in the PIM language.

- For each problem we must find a solution, express it in the PSM language, and include it in the RHS of the corresponding model type mapping rule.

I said that this solution is theoretical because it is impossible in practice. Each platform can solve an infinite number of problems. The best we can do is to identify the most important (most widely used) of them. It follows that we can't construct a model type mapping rule for each concept from the PSM language. Therefore, model type mapping rules can't be used as the only means for combining PIMs and PSMs.

Where model type mapping rules can't be used because of the high level of abstraction of the PIM language, we can use *model instance mapping rules*. Model instance mapping rules allow to transform model elements from the PIM in a user-defined way. They don't depend on how detailed the PIM language is because they can map *any* PIM element to *any* concept (or pattern) from the PSM language. An architect can choose a PIM element which should be mapped in a special way and assign it a mark. The mark is a name of the model instance mapping rule which should be used. Of course, each mapping based on a model instance mapping rule must make sense and it is the architects responsibility to ensure that each mark is assigned to a proper PIM element. Model instance mapping rules can be used with any PIM language but they still require a highly detailed PSM language. Otherwise, model instance mapping rules will suffer from the same problem as the model type mapping rules — it will be impossible to take advantage of all strengths of the platform.

Model instance mapping rules are used together with model type mapping rules. How many model instance mapping rules a particular PIM-to-PSM mapping may require depends on the PIM language. On one extreme, if the PIM language is at the same level of abstraction as the PSM language, then there would be no need for model instance mapping rules at all. On the other, if the PIM language is very abstract, then most of the mapping rules will be model instance mapping rules. In fact, the entire PSM can be thought of as a RHS of a model instance mapping rule where the LHS is a PIM consisting of a single element. It is obvious that the more model instance mapping rules we use, the less valuable combined PIM-PSM becomes. The whole idea of combining a PIM with PSMs is to achieve an automatic synchronization between a PIM and all associated PSMs. Changes to all models should be made only one place — the combined PIM-PSM. This is difficult to achieve with model instance mapping rules. Because they express mappings in terms of PIM elements and not in terms of the concepts from the PIM language, they may need to be redefined every time we update a PIM.

Chapter 8

Conclusions and future work

In this chapter I will try to give an answer to the main questions of this master thesis:

- *Is it possible to combine a PIM and the corresponding PSMs in one and the same model?*
- *If yes — how can this be done?*
- *If no — what makes it difficult?*
- *In either cases — what future work in this direction can be done?*

Why do we need combined PIM-PSM?

In the introductory chapter I showed that current software projects tend to be very big and complicated and that we need a new approach to software development to deal with this complexity. Many people believe that modeling is the new step in the evolution of the software development. The aim of the MDA initiative is to develop a standard software development methodology where models play the central role. The MDA proposes the use of models through the entire cycle of the software development project as the main tools for analyzing system requirements, developing system architecture in a platform-independent way, and porting platform-independent solution to a platform of choice.

We have also seen in the introductory chapter that a successful software development project depends on the three viability variables: the cost of production, the quality of the software, and the longevity of the software. If modeling is really a new step in the software development evolution then it must increase the quality all three viability variables. The quality of the software can be certainly improved if we use modeling because it helps to illustrate the problem visually and analyze it from different viewpoints. However, it is not so obvious that modeling can improve the two other viability variables: the cost of production and the longevity of the software. Only after a careful investigation of the MDA we can see whether it holds or not.

The answer to the last question depends on many aspects of the MDA. My research of the MDA was not so overwhelming to cover absolutely all of them.

However, one of the most important aspects — the possibility of combining PIMs and PSMs — was the main question of this master thesis. If we can make combined PIM-PSM then the quality of the viability variables can also be improved because:

- Any MDA-compliant architecture development methodology requires a large amount of models. Maintaining all of them is a difficult task. A combined PIM-PSM allows to hold several models in one model. This can spare much time because updating several models can be done simultaneously. Thus, the cost of production can be reduced.
- Combined PIM-PSM is only possible if there exist PIM-to-PSM mapping rules that can be processed automatically. Such rules allow automatic porting of the application to any platform. Thus, the longevity of the application can be increased.

Combined PIM-PSM: possible or not?

So, the question that I am about to answer is: Is it really possible to combine a PIM and its PSMs? Surprisingly, my answer is both *yes* and *no*. Here are my arguments.

It is *no* because it is generally impossible to make a PIM language that is at the same level of abstraction as a PSM language. PIMs with such level of abstraction are necessary for constructing unambiguous model type mapping rules which can use all concepts from the platform language. If the PIM language is at a higher level of abstraction than the platform language then many of the concepts specific to the platform will be not covered by the mapping rules. Using such mapping rules can lead to a PSM of a bad quality, i.e. the PSM will describe an inefficient solution to the problem expressed in the PIM. Although this solution will be workable, it will never be chosen by an architect who does not follow the PIM/PSM approach.

It is *yes* because despite of the fact that no automatic synchronization between an arbitrary PIM and arbitrary PSM is possible, we can come very close to it:

- In some special cases model type mapping rules may lead to an optimal PSM. A PIM language may lack some advanced concepts, but it usually covers the basic ones. Therefore, if we have a PIM consisting only of simple concepts (e.g. classes without any stereotypes) then there are big chances that we have model type mapping rules which map these concepts.
- Sometimes we can accept a PSM which is not an optimal solution for the problem from the PIM. Such PSMs can be accepted for small and non-critical applications where performance issues are not the first priority.
- Whenever we can not use model type mapping rules, we can use model instance mapping rules. It is difficult to achieve a full PIM-PSM synchronization when we use model instance mapping rules, therefore, they must be used only when absolutely necessary. If the PIM language has a low level of abstraction (although not the same as the level of abstraction of the platform language), then the use of model instance mapping rules

will be minimized. Therefore, the amount of manual work to synchronize model will also be minimized.

Future work

The MDA is a relatively young initiative. Now we are just at the beginning of a long process of standardizing and adoption of its many concepts. A lot of work has to be done before we can begin to use combined PIM-PSM. Among the most important issues are:

- *Highly detailed and standardized PIM languages for different markets and technologies.* We saw that the more detailed a PIM language is, the more precise the model type mapping rules will be. It will raise the degree of automatic synchronization. Standardization of PIM languages may lead to standardization of mapping rules.

- *Highly detailed and standardized languages for different platforms.* Platform languages should describe not only basic features such as type systems but also the best practices of using these platforms (patterns). Standardizing of platform languages is also a requirement for standardized mapping rules.

- *Predefined and standardized model type mapping rules that cover all important aspects from PIM languages and PSM languages.* Standardizing mapping rules allows reuse of mapping rules. Thus, it would be unnecessary to define own set of mapping rules for each PIM.

- *Standard language for expressing mapping rules and the possibility to process mapping rules automatically.* In this master thesis I paid very little attention to the question of expressing of mapping rules. I only showed that no common solution exists. However, it is a question of critical importance. If we can not make a transformer that processes mapping rules automatically, then the whole idea of combined PIM-PSM is almost worthless.

List of Figures

1.1 The Java Pet Store and the .NET Pet Shop 15
1.2 Use Case Diagram for the Pet Shop e-commerce Web application 16

2.1 Model Driven Architecture 19
2.2 Using MOF to define a simple meta-model 21
2.3 MOF meta-levels 21
2.4 PIM and PSM relations 26
2.5 Graphical notation of a concept from UML diagram 30
2.6 Dependency relationship 30
2.7 Association relationship 31
2.8 Generalization relationship 31
2.9 Stereotypes and tagged values 32
2.10 Stereotype specification 33

3.1 Defining primitive types 37
3.2 Navigable end name clashing 39
3.3 Pet Shop modules 40
3.4 Customer module PIM 43
3.5 Catalog module PIM 45
3.6 Order module PIM 48

4.1 J2EE Platform overview 51
4.2 .NET Platform overview 52
4.3 EJB model 62
4.4 Summary of J2EE and .NET enterprise architectures 67

5.1 Use of model transformations in MDA 71
5.2 General prinsiples of model transformations 73
5.3 Example of relation syntax 74

6.1 MDA transformation 83
6.2 Three kinds of mappings 84
6.3 Class mapping 86
6.4 Mapping of collections 87
6.5 PIM-to-J2EE model type mapping of «Entity» class 103
6.6 Example of PIM-to-J2EE model type mapping of PIM «Entity» class 104
6.7 PIM-to-J2EE model type mapping of PIM «Entity» class with missing «unique» attribute 105

6.8 Example of PIM-to-J2EE model type mapping of PIM «Entity» class with missing «unique» attribute 106
6.9 PIM-to-J2EE model type mapping of PIM «Controller» class . . 107
6.10 Example of PIM-to-J2EE model type mapping of PIM «Controller» class 108
6.11 PIM-to-.NET model type mapping of PIM «Controller» and «Entity» classes 109
6.12 Example of PIM-to-.NET model type mapping of PIM «Controller» and «Entity» classes 110
6.13 PIM-to-.NET model type mapping of PIM «Controller» class that supports transactions 111
6.14 Example of PIM-to-.NET model type mapping of PIM «Controller» class that supports transactions 111
6.15 PIM-to-J2EE model instance mapping of a PIM element marked with *J2EE:DAO* mark 112
6.16 Example of PIM-to-J2EE model instance mapping of a PIM element marked with *J2EE:DAO* mark 113
6.17 PIM-to-.NET instance type mapping of a PIM element marked with *.NET:Remove* mark 114
6.18 Example of PIM-to-.NET model instance mapping of a PIM element marked with *.NET:Remove* mark 114
6.19 PIM-to-.NET instance type mapping of a PIM element marked with *.NET:addFunction()* mark 115
6.20 Example of PIM-to-.NET model instance mapping of a PIM element marked with *.NET:addFunction()* mark 115

7.1 Model Driven Architecture 119

List of Tables

4.1 Comparing Runtime Environment of J2EE and .NET 54
4.2 Inheritance modifiers of Java and C# 57
4.3 Access modifiers of Java and C# 57
4.4 Example of base class libraries of Java and C# 59
4.5 Component technologies in J2EE and .NET 63

6.1 Mapping of basic data types from PIM to J2EE and .NET 86
6.2 Truth table for PIM-to-J2EE and PIM-to-.NET Collections Mapping 88

Glossary

ADO.NET The .NET API for accessing data sources.

ADSI (Active Directory Server Interface) A Microsoft technology that abstracts the capabilities of directory services from different network providers to present a single set of directory service interfaces for accessing and managing network resources.

BMP (Bean Managed Persistence) A way to persist entity beans when all necessary persistence logic is implemented by the programmer inside the entity bean.

C# A new C++ derivative programming language that is similar in functionality, look, and feel to Java.

CIM (Computational Independent Model) The base PIM that expresses only business functionality and behavior.

COBOL One of the primary business programming languages.

COM+ The .NET middle tier infrastructure designed to support business components.

CLR (Common Language Runtime) The .NET runtime that provides support for compiled MSIL.

CLS (Common Language Specification) CLS is a subset of CTS. Components that conform to CLS are guaranteed to be usable by any other component that conforms to the specification.

CMP (Container Managed Persistence) A way to persist entity beans when all necessary persistence logic is implemented by the container.

CTS (Common Type System) The CTS defines how .NET types are declared, used, and managed at run time.

EJB (Enterprise Java Beans) The J2EE middle tier infrastructure designed to support business components.

Entity beans Persistent objects that can be stored in permanent storage.

HTML The industry standard for describing browser displays.

HTTP The industry standard for communications over Internet.

JAAS (Java Authentication and Authorization Service) A set of J2EE APIs that enable services to authenticate and enforce access controls upon users.

J2EE (Java 2 Enterprise Edition) A platform-independent, Java-centric environment from Sun for developing, building and deploying Web-based enterprise applications online.

J2SE (Java 2 Standard Edition) Java programming language's core API set.

Java Programming language associated with J2EE.

Java Applets A packaging technology for downloading and running Java code on the client side in a browser.

Java byte code The intermediary language run within JVM.

Java Connectors The J2EE API used for connecting to EIS.

JSP (Java Server Pages) The J2EE technology for presentation tier.

Java Servlets The J2EE technology for presentation tier.

JDO (Java Data Objects) An architecture that provides a standard way to transparently persist plain Java objects.

JDBC (Java Database Connectivity) The J2EE API for accessing databases.

JMS (Java Messaging Service) The J2EE API for accessing message queues.

JNDI (Java Naming and Directory Interface) The API for naming and locating specific instances used in J2EE.

JTS (Java Transaction Server) The J2EE API used for managing transactions boundaries.

JVM (Java Virtual Machine) The Java language environment.

HIS (Microsoft Host Integration Server) A platform for connecting to legacy, UNIX, and mainframe platforms.

MDA (Model Driven Architecture) An approach to IT system specification that separates the specification of functionality from the specification of the implementation of that functionality on a specific technology platform.

Model A formal specification of the function, structure, and/or behavior of an application system.

MS-DTC (MS Distributed Transaction Coordinator) The .NET technology that manages the two phase commit protocol used in distribution transaction coordination.

MSIL (MS Intermediate Language) Intermediate code produced by CLR.

MTS (MS Transaction Server) The original middle tier, component based infrastructure designed to support highly scalable applications.

OCL (Object Constraint Language) A formal language for describing constraints about objects in UML models.

OMG (Object Management Group) Industry consortium which goal is to provide a common framework for developing applications using object-oriented programming techniques

.NET A Windows-based environment from Microsoft for developing, building and deploying enterprise applications and Web Services.

PIM (Platform Independent Model) A model of a subsystem that contains no information specific to the platform, or the technology that is used to realize it.

Platform A set of subsystems/technologies that provide a coherent set of functionality through interfaces and specified usage patterns that any subsystem that depends on the platform can use without concern for the details of how the functionality provided by the platform is implemented.

PSM (Platform Specific Model) A model of a subsystem that includes information about the specific technology that is used in the realization of it on a specific platform, and hence possibly contains elements that are specific to the platform.

RMI/IIOP (Remote Method Invocation over Internet InterOrb Protocol) The protocol used for communicating with components in J2EE based on the CORBA protocol.

SOAP (Simple Object Access Protocol) An industry standard for packaging of method requests.

UDDI (Universal Description, Discovery, and Integration) An industry standard for publishing web services.

UML (Unified Modeling Language) An OMG standard language for specifying the structure and behavior of systems. The standard defines an abstract syntax and a graphical concrete syntax.

UML Profile A definition of a set of stereotypes, tagged values and constraints that extend elements of the UML metamodel.

UML Profile for Enterprise Application Integration (EAI) Provides a metadata interchange standard for information about accessing application interfaces. The goal is to simplify application integration by standardizing application metadata for invoking and translating application information.

UML Profile for Enterprise Distributed Object Computing (EDOC) The vision of the EDOC Profile is to simplify the development of component based EDOC systems by means of a modeling framework, based on UML 1.4 and conforming to the OMG Model Driven Architecture. EDOC

is composed of seven specifications: (1) Enterprise Collaboration Architecture (ECA); (2) Metamodel and UML Profile for Java and EJB; (3) Flow Composition Model (FCM); (4) UML Profile for Patterns; (5) UML Profile for ECA; (6) UML Profile for Meta Object Facility; and (7) UML Profile for Relationships.

XML (Extensible Markup Language) An industry standard that enables the definition, transmission, validation, and interpretation of data between applications and between organizations.

XMI (XML Metadata Interchange) An OMG standard that facilitates interchange of models via XML documents.

XSLT (eXtensible Stylesheet Language Transformation) A language for transforming XML documents into other XML documents.

XPath A language for addressing parts of an XML document, designed to be used by XSLT.

Bibliography

[1] J. Araújo. Integration and Transformation of UML Models. *Object Oriented Technology, ECOOP 2002 Workshop Reader*, 2002.

[2] Jean Bézivin and Sébastien Gérard. A Preliminary Identification of MDA Components.
http://www.softmetaware.com/oopsla2002/mda-workshop.html.

[3] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[4] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches, 2003. Available at http:
www.omg.org.

[5] David S. Frankel. *Model Driven Architecture*. Wiley Publishing, 2003.

[6] Jan Hendrik Hausmann and Stuart Kent. Visualizing Model Mappings in UML. *Proceedings of the 2003 ACM symposium on Software visualization*, 2003.

[7] W. Ho, F. Pennaneach, and N. Plouzeau. Umlaut: A Framework for Weaving UML-Based Aspect-Oriented Designs. *Proceedings of the 33rd International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 33)*, 2000.

[8] Java Pet Store Sample Application 1.3.2.
http://java.sun.com/blueprints/code/index.html.

[9] Java 2 Platform, Enterprise Edition Home Page.
http://java.sun.com/j2ee/.

[10] Java Technology Home Page.
http://java.sun.com/.

[11] Raghhild Kobro Runde and Ketil Stolen. What is Model Driven Architecture?
University of Oslo, Department of Informatics, 2003.

[12] J. Kovse and T Harder. Generic XMI-Based UML Model Transformations. *Proceedings of the 8th International Conference on Object-Oriented Information Systems*, 2002.

[13] Joaquin Miller and Jishnu Mukerji. MDA Guide (Draft Version 0.2). http://www.omg.org, 2003.

[14] Microsoft .NET Home Page. http://www.microsoft.com/net.

[15] Microsoft .NET Pet Shop 3.0. http://msdn.microsoft.com/library/en-us/dnbda/html/bdasamppet3.asp.

[16] Jeffrey Ritcher. *Applied Microsoft .NET Framework Programming*. Microsoft Press, 2002.

[17] Ed Roman, Scott W. Ambler, and Tyler Jewell. *Mastering Enterprise JavaBeans*. Wiley Publishing, 2002.

[18] Roger Sessions. Java 2 Enterprise Edition (J2EE) versus The .NET Platform.
Available at http://www.objectwatch.com, 2001.

[19] Jim Stone. Java/J2EE versus C#/.NET.
Available at http://www.theobjectcenter.com.

[20] XSL Transformations (XSLT). Version 1.0. http://www.w3.org/TR/xslt.